I Love Ruby

By Karthikeyan A K

Contents

I love Ruby

Ruby is easy to learn programming language, it was invented by a guy named Matz in Japan. Ruby is a free software and can be used by any one for no cost. Ruby's popularity was initially confined to Japan, later it slowly trickled out to rest of the world. Things changed with the emergence of Ruby on Rails which is a popular web-development framework thats written with Ruby.

I was thrilled when I started to program in Ruby. One of the first application that I created was a student ranking software for my mom who is a teacher. I was able to write the console based application in just 32 lines. This opened my eyes and made me realize the power of Ruby. The language was simple, easy to learn and nearly perfect. Currently I am an amateur Ruby on Rails programmer.

This book is written for Ubuntu GNU Linux users, its because I think Ubuntu will conquer desktops of programmers in near future. Almost all who have Debian Linux based distro should feel at home while trying to learn Ruby using this book. If you are using other operating systems like Solaris, OSX or Windows please contact your Operating System help channels to learn how to install or get started with Ruby. You can also visit http://ruby-lang.org to learn and how get started with Ruby.

Copyright

I

Hi, I am A.K.Karthikeyan, the author of this book. I came across Ruby and found that the language was dazzling. I thought why not write a book for self study and so was created this book.

Currently I am an web programmer, I use Ruby on Rails due to its great innovative features. You can contact me at mindaslab@gmail.com or tweet to @karthik_ak

This book is no match to other Ruby best sellers, this is just an attempt to have a personalized Ruby study material.

Kannan Doss

I am very much grateful Kannan Doss who works in Mind As Lab as a apprentice web developer and a Ruby on Rails Geek for putting sincere efforts to improve the quality of this book. He has proof read the entire book, spotted mistakes he could find, cataloged them and corrected them.

Without him the book wont be a good quality as it is now. I am not saying that this book is superb. But his efforts truly played a decisive role to make this book what it is.

You can contact Kannan at doss.kannan@gmail.com

Getting this book

I am not quiet sure where to host this book right now, weather on dropbox or lulu which helps me keep count of number copies been downloaded. All annoincements about this book are availabe in facebook in this URL: https://www.facebook.com/pages/I-love-ruby/172269549451705 , there is no necessity for you to have facebook account to access this page.

Getting example programs

The example programs in this book are hosted in Github in the following URL: https://github.com/mindaslab/ilrx . I realize many who read this book are starters, who have just plunged into programming. Hence ou can download all wexamples by clicking this link: https://github.com/mindaslab/ilrx/zipball/master

Installing Ruby

Installing Ruby on Ubuntu GNU/Linux

Okay, you need to install a thing called RVM (ruby version manager) which will control which version you are using. Why? Its because ruby version changes fast. Before you had 1.8, now 1.9 and soon ruby 2 will be out. Apart from just using ruby, you will also use it for other stuff like web development like Sinatra and Ruby on Rails. You might need to change from one version to other without uninstalling and reinstalling ruby again and again. RVM manages this for you. With simple commands we can switch between Ruby versions easily.

Installing RVM :

OK, to install RVM, you needto make sure you have the necessary libraries to compile and install it. So copy the stuff below, (omitting the starting $) and paste it into terminal

If you are using the PDF version of this book, copying and pasting these codes will result in an error, so I have put all the codes in this file *ruby_install.bash* , use it to avoid tedious typing

```
$ sudo apt-get install build-essential bison openssl libreadline6 libreadline6-dev curl git-core zlib1g zlib1g-dev libssl-dev libyaml-dev libsqlite3-0 libsqlite3-dev sqlite3 libxml2-dev libxslt-dev autoconf libc6-dev ncurses-dev automake
```

Now install RVM using the following command

```
$ curl -L https://get.rvm.io | bash -s stable --ruby
```

Once done give these commands into terminal. These will tell Ubuntu GNU / Linux where to find the ruby interpreter.

```
$ echo '[[ -s "$HOME/.rvm/scripts/rvm" ]] && . "$HOME/.rvm/scripts/rvm" # Load RVM function' >> ~/.bashrc


$ source ~/.bashrc
```

Once done, possibly restart computer and in terminal type the following

```
$ ruby -v
```

It will spit an output somethin like this

```
ruby 1.9.3p194 (2012-04-20 revision 35410) [x86_64-linux]
```

Then all is OK!

Installing on Windows

WHAT? U think I am insane to use Windows now?

Installing on Mac

Beauty is just skin deep. Only stupids will get seduced by it. If you are using mac, break free from it and try out GNU/Linux which is rock solid and DRM free.

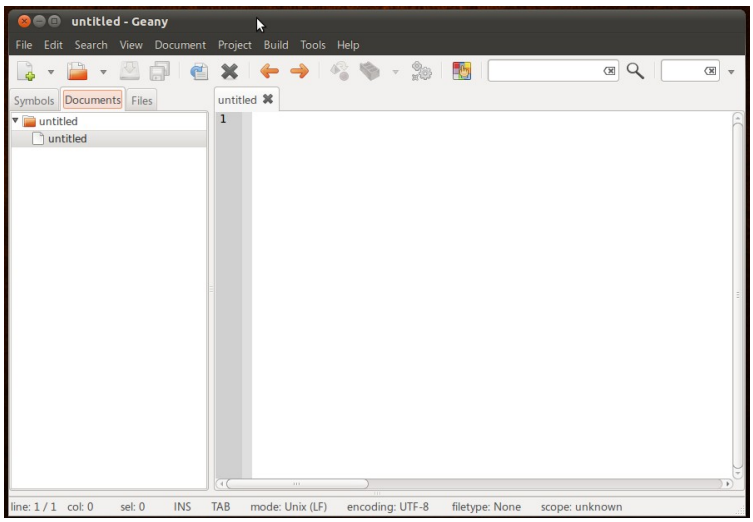
Installing IDE

You need a good IDE (Integrated development environment) to get started with Ruby. I recommend simple and light weight IDE Geany[1]. In Ubuntu machine just type

```
sudo apt-get install geany
```

for the IDE to get installed. If the system asks for administrator password, provide it.

Click the dash button and type in geany. Click on the geany icon, you will get it :-)



1 Windows users goto http://www.geany.org/ to download the installer

Online Resources

Ruby has got a excellent online community of hackers who are ready to help almost any one who has almost any doubt about Ruby. They love the programming language and want others to love and experience it too. Ruby is a great programming language that will put something good in your heart. Once you have learned it and start to interact with fellow hackers, you will naturally tend to help others. So do visit the websites recommended in this section. They might be of great use to you.

Ruby Website

URL: http://ruby-lang.org

Ruby website is a great place to start with Ruby. It provides you with the installers to install Ruby on your operating system. It has cool links like 'Try Ruby! in your browser', which lets you try out Ruby right from your web browser and a link called 'Ruby in Twenty Minutes' teaches you basics of Ruby programming. Ruby is such a simple language that you just need 20 minutes to grasp it! Trust me its true!

Ruby Forum

URL: http://www.ruby-forum.com/

So where to go if you have doubts with Ruby? You can visit http://www.ruby-forum.com/ which is a website thats nothing but collection of Ruby forums and contains lot of question and answers about Ruby. No matter how stupid it may be, you can post your question. A kind enough gentle man (or a lady if you are lucky) will answer it. For the sake of heaven, never fail to ask questions. The difference between a good and great programmer could be just a question you ask.

Twitter

URL: http://twitter.com

Twitter is a socializing website. Then why on Earth am I putting it here. Well lot of Ruby programmers use twitter, possibly because its written with Ruby. To know the latest news about “Ruby Programming”, type it in the search bar and press search. You will get latest trending topics

about Ruby language.

The screenshot below shows my search for Ruby programming. Try searches like “Ruby language” and blah blah....

Getting Started

Having installed the needed software, lets gets started.

Interactive Ruby

Ruby provides us a easy way to interact with it, this feature is called interactive ruby or irb. With irb you can type small bits of ruby code in your console and see it get executed. irb is a great tool to check out small pieces of Ruby code. In your terminal type `irb` or `irb --simple-prompt` , you will be getting prompt as shown

```
irb(main):001:0>
```

The above prompt will be got if you had typed `irb`

```
>>
```

The above prompt will be got if you had typed `irb --simple-prompt`, in examples from now on I will be using the simple prompt as its simple for me to write in this book. Lets write our first hello world program, in the prompt type the following (don't type those `>>`)

```
>> puts 'Hello World!'
```

When you press enter, you will get output as follows. In Ruby `puts` is used for printing some thing onto the console.

```
Hello World !
=> nil
```

Very well, we have completed our hello world program under a minute. Lets check what is 56 to the power of 31

```
>> 56**31
=> 1562531701075863192779448904272185314811647640213651456
```

OOPS! You never thought it would be such a large number, did you? Any way, the `**` is used to find a number raised to the power of another number.

To quit irb and return to normal console or terminal prompt type `quit`

Doing some Math

Computer is a device that computes, or does some math. With `irb` we can do easy math. If you

don't like to work with numbers, ruby can do it for you. So, first, lets add these numbers : 1, 45, 67, 893,72, 56 and -128. To do so in your *irb* prompt just type these numbers separated by a plus '+' sign and you will get the result

```
>> 1 +  45 + 67 + 893 + 72 + 56 + -128
=> 1006
```

Here are some common math operators that you will find useful

Operator	What they do
+	Adds numbers
-	Subtracts a number from another number
/	Divides a number with another number
*	Multiplies two numbers
**	Finds a number raised to the power of another
%	Finds the remainder
+=	Adds and assigns a value to a variable
-=	Subtracts and assigns a value to a variable
*=	Multiply and assigns a value to a variable
/=	Divides and assigns a value to a variable
%=	Finds the remainder and assigns it to a variable

Addition Example: Lets say that I want to add 56 and 72 and find its result, I can do it as shown:

```
>> 56+72
=> 128
```

Subtraction Example: In this example I am subtracting 64 from 112

```
>> 112-64
=> 48
```

Division Example: Lets say I want to divide 117 by12 and find the quotient, I can do in Ruby like this

```
>> 117/12
=> 9
```

Power Example: Lets say I want to find what we will get by cubing five (five raised to the power of three), I can do it in Ruby as shown

```
>> 5**3
=> 125
```

Modulus or Remainder Example: I want to know what we will get as remainder when we divide 21 by 4, I can do it as shown

```
>> 21%4
=> 1
```

Addition with assignment Example: Lets declare a variable `i`, set it to zero and add 24 to it. In ruby you can do it as shown

```
>> i = 0
=> 0
>> i+=24
=> 24
>> i
=> 24
```

At the end when we type i and see we get 24. This means `i` holds the value 24 in it.

Subtraction with assignment Example: Lets declare a variable `j` , assign it with a value 50 and take away 17 from it

```
>> j = 50
=> 50
>> j -= 17
=> 33
>> j
=> 33
```

At the end when we type `j` and see we get 33. This means `j` holds the value 33 in it.

Multiplication with assignment Example: Lets declare a variable `k`, set it to 3 and multiply it by nine

```
>> k = 3
=> 3
>> k *= 9
=> 27
>> k
=> 27
```

At the end when we type `k` and see we get 27. This means `k` holds the value 27 in it.

Division with assignment Example: Lets declare a variable `s`, set it to 25 and divide it by 5

```
>> s = 25
=> 25
>> s /= 5
=> 5
>> s
=> 5
```

At the end when we type `s` and see we get 5. This means `s` holds the value 5 in it.

Try other operators on your own, I'm running out of patience.

Space doesn't matter

Lets say that I want to add 54 with 62, how can I command `irb` to do it. Should it be `54+62` or can I

leave spaces so that code could be neatly written like `54 + 62` . Well fortunately in Ruby leaving spaces doesn't really matter you can give it in any number of ways as shown below and still get the same result.

```
>> 54+62
=> 116
>> 54 +62
=> 116
>> 54+ 62
=> 116
>> 54 + 62
=> 116
>> 54           +       62
=> 116
```

Notice that the plus weather it sticks with 54 or 62 or has space between them, no matter how long the space is, it prints out the right result.

Decimals

When you divide 5 by 3 in ruby you get result as follows

```
>> 5/3
=> 1
```

In other words it gives the quotient. In reality 5 divided by 3 is almost 1.666666666666666667, so how to get this answer? The truth is 5 and 3 are integers, or numbers that don't have decimal part. If you want a fairly accurate answer you can rephrase your command to Ruby as follows

```
>> 5.0/3
=> 1.66666666666667
```

In the above way, we are specifying 5.0 instead of 5, in other words we are forcing Ruby to make a floating point or decimal calculation instead of integer calculation. This makes Ruby to give an fairly accurate answer.

Variables

Variables are something that stores value in it. You can imagine them as a box which can hold pebbles. If a box named `a` holds five pebbles then its value is 5, if another box `b` holds 3 pebbles, then its value is three. Let say you get an new box called `c` and you want its value to be the sum of box `a` and box `b`, then you simply add number of pebbles in `a` and `b`, it totals to 8, you put 8 pebbles in c to make `c = a+b`. I hope you have got a hint what a variable is. Lets program it in Ruby

```
>> a = 5
=> 5
>> b = 3
=> 3
```

```
>> c = a+b
=> 8
```

Lets try another problem, I buy 50 mangoes from a farmer at Rs 10/- and bring it to the market and sell it at Rs 15/- each, what is my profit.

Answer:

OK first I have 50 mangoes so in *irb* I type as follows:

```
>> mangoes = 50
=> 50
```

So I have assigned the value of *50* to a variable *mangoes*. Next I declare and assign a value of *10* to a variable *buy_price* as shown:

```
>> buy_price = 10
=> 10
```

In a similar fashion I assign 15 to a variable named *sell_price*

```
>> sell_price = 15
=> 15
```

Now profit per mango is the difference between sell and buy price, hence I can calculate it as shown

```
>> profit = sell_price - buy_price
=> 5
```

By selling a mango I get a profit of Rs 5/-, what will I get by selling 50 mangoes? Its a multiple of *profit* with *mangoes* and we get it as shown

```
>> total_profit = profit * mangoes
=> 250
```

So by selling 50 mangoes we can earn a profit of Rs 250/-. Lets say that we have bought 72 mangoes, now we want to know what profit would be, this can be easily done by changing or varying the value *mangoes* from 50 to 72 and recalculating the *total_profit* as shown below

```
>> mangoes = 72
>> total_profit = profit * mangoes
=> 360
```

Now you may know why we call these things are variables, a variable is a box that can contain any value it wants. Just like you can add or take away pebbles from a box, you can do the same to variables.

Naming Convention

In the mango example, you would have noticed that I have given the names of variables as *buy_price*, *sell_price*, *total_profit* and not as *buy price*, *sell price*, *total profit*, why so? It turns out that one must follow a certain naming convention or rules when naming a variable. The rules of naming a variable are as follows

1. There must be no space in between variable names
2. There must be no special character except underscore _ in a variable name
3. A variable name can have numbers
    1. A variable name must not start with a number
4. A variable must either start with a character or an underscore
    1. Capital character should not appear at the start of variable

Below given are examples of valid variable names

```
mango
total_price
mango_
_mango
buyPrice
boeing747
boeing_747
iam23yrsold
```

Below are given examples of invalid variable names

```
34signals
Mango
total cost
```

The underscore – a special variable

Suppose we want to find whats 87 raised to the power 12, we can do as follows

```
>> 87**12
=> 188031682201497672618081
```

Now we want to multiply the result with 5 and see the answer, now the above result is a whopy 24 digit number and we must type all of it and put a star five to get an answer, thats a lot of work! If you are a programmer, laziness should flow in your veins otherwise find another profession. One way is to assign this value to a variable and multiply it by 5 as shown below

```
>> a = 87 ** 12
=> 188031682201497672618081
```

```
>> a*5
=> 940158411007488363090405
```

However there is another easy way as shown below

```
>> 87**12
=> 188031682201497672618081
>> _*5
=> 940158411007488363090405
```

I did find out 87 raised to the power of 12, and after that I multiplies underscore _ with five! But how come? Underscore is a special kind of variable, in it the result of last execution gets stored automatically. If you want to use the last obtained output you can do so by using underscore _ as a variable[2].

Constants

Unlike variables, some values must be constant, for example the radius of the Earth is constant, the speed of light is constant. In problems that deal with these kind of issues, or in situations where you are absolutely certain that some values wont change, you can use constants.

A constant can be thought as a variable who's value doesn't change. Constants in Ruby starts with a capital letter, it could then be followed by alphabets, numbers and underscore. Lets now have a constant called Pi who value will be equal to mathematical pi π , to do so just type the following in irb prompt

```
>> Pi = 3.1428
=> 3.1428
```

Having assigned the value of π to a constant named Pi, we will now try to find area a circle whose radius is 7 units, so lets use our faithful calculator the irb. We know that radius of a circle is πr^2 [3], where r is the circles radius. In your irb prompt we can do the calculation as follows

```
>> r =  7
=> 7
>> Pi*r**2
=> 153.9972
```

So we find area of circle is roughly 153.9972 square units, which is very near to the exact value of 154 square units.

One can ask weather can we change value of constant? I don't say its impossible, but if we change ruby gives us warning that we are changing the value of a constant, after the warning the constant gets changed anyway.

2 This underscore as a variable works only in interactive ruby (irb). When you are executing a ruby program typed in a file this wonk work. See section Will underscore work in a Ruby file?

3 Well I knew it because Albert Einstien is my friend. I just talked to him and he told me the formula for area of circle

```
>> Pi=5
(irb):35: warning: already initialized constant Pi
=> 5
```

In the above example I have re assigned the value of Pi to 5, as you can see in the second line, Ruby interpreter does throws out a warning that Pi is already initialized constant, but any way the value of Pi gets changed to 5. It is strongly discouraged not to change values of constants in professional programming.

Strings

Till now we have seen about numbers, now lets see something about text. In computers text are called as string[4]. OK lets see about strings in Ruby. Lets start with an hello world. In your irb type hello world as shown

```
>> "hello world"
=> "hello world"
```

As a response you get an *"hello world"*. In short, string is any stuff thats surrounded by " or by '

Now lets try the above example by surrounding the above hello world with single quotes

```
>> 'hello world'
=> "hello world"
```

Well you do get the same response. So whats the difference between single and double quotes? Take a look at the following example

```
>> time_now = Time.new # Get the current time into a variable
=> Fri Jan 15 16:43:31 +0530 2010
>> "Hello world, the time is now #{time_now}"
=> "Hello world, the time is now Fri Jan 15 16:43:31 +0530 2010"
>> 'Hello world, the time is now #{time_now}'
=> "Hello world, the time is now \#{time_now}"
```

At first we declare a variable called *time_now* and store the current time into it. The current time in Ruby is got by *Time.new* command. Now we have a variable and we can embed it into a string by putting it like *#{put_your_variable_here}*. So we want to tell the world the time now is something, so we give a command as shown

```
>> "Hello world, the time is now #{time_now}"
=> "Hello world, the time is now Fri Jan 15 16:43:31 +0530 2010"
```

and we get a proper result. Note that you have enclosed the string with a double quotes. Now lets try the same thing with single quotes

```
>> 'Hello world, the time is now #{time_now}'
=> "Hello world, the time is now \#{time_now}"
```

We see that in this case the world is not able to see what time it is, rather its able to see a ugly string

4 Possibly because they are string of characters

as shown

```
"Hello world, the time is now \#{time_now}"
```

What ever thats put between single quotes gets printed as it is. You might ask why `#` is printed as `\#`, well we will see it in escape sequence soon.

String Functions

There are certain cool things you can do with a string with the built in functions and routines packed into Ruby. For example if I want to find the length of a string I can use the length function as shown

```
>> "my name is billa".length
=> 16
```

There are many functions, some of which are given in the table shown. I must warn you that this table is not comprehensive, you must check the Ruby documentation[5] for a comprehensive coverage.

Input	Output	Notes
`"my name is billa".length`	`16`	The *`length`* function finds the length of a string
`"my name is billa".reverse`	`allib si eman ym`	The *`reverse`* function reverses a string
`"my name is billa".capitalize`	`My name is billa`	Capitalizes the given string
`"my name is billa".upcase`	`MY NAME IS BILLA`	Converts lower case characters to uppercase
`"MY NAME IS BILLA".downcase`	`my name is billa`	Converts uppercase characters to lower case
`"my name is billa".next`	`my name is billb`	This is quiet illogical function that prints the next logical String
`"my name is billa".empty?`	`false`	Returns *`true`* if string is empty, else returns *`false`*
`"".empty?`	`TRUE`	Returns *`true`* if string is empty, else returns *`false`*

OK, so we have seen some functions, lets now see what operations can be performed on string. The first one is concatenation in which two or more strings can be joined together, take a look at

5

example below

```
>> "Hello"+" "+"World!"
=> "Hello World!"
```

In the code above, I have joined three strings “Hello' a (space) “ “and “World!” using a plus sign, the same operation can be done with string variables too as shown below

```
>> string_1 = "Hello"
=> "Hello"
>> string_2 = "World!"
=> "World!"
>> string_1+" "+string_2
=> "Hello World!"
```

OK now, we have studied a lot, a bit of meditation will help, lets chant OM[6] to cleanse and reset our mind. You know Ruby can meditate for you? In your *irb* type the following

```
>> "OM "*10
```

For heaven sake don't type >> ! And here is your result

```
=> "OM OM OM OM OM OM OM OM OM OM "
```

The multiplication operator followed by a number prints a string N number of times, where N is the number given after *.

Escape sequence

Whenever you type a statement like *puts "Hello World!"* the Ruby interpreter prints *Hello World!*. That is every thing between “ and “ gets printed. Well not always. There are some things that you can put between “ and “ that will escape the normal printing sequence. Launch your irb and type the example given below:

```
>> puts "Hello \r World!"
 World!
=> nil
```

Surprise, you see only *World!* getting printed. What happened to the Hello? Well the *\r* character stands for carriage return, which means the Hello does get printed. Then the carriage/cursor returns to the beginning of the line and World! gets over written on it. Like *\r* stands for carriage return, *\n* stands for newline. Type the example below in irb

```
>> puts "Hello \n World!"
Hello
 World!
=> nil
```

As you can see *Hello* gets printed in first line and *World!* gets printed in next. This is because we have placed a new line character *\n* in between them.

6 A magical word uttered by saints in India

Well now lets take a scenario, we now know that `\r`, `\n` and possibly others are non printing characters. Now how to print `\n` or `\r` in our output. As it turns out that putting a double backward slash would print a backward slash in output as demonstrated by example below.

```
>> puts "Hello \\n World! => Hello \n World!"
Hello \n World! => Hello
 World!
=> nil
```

In a similar fashion `\t` puts tab spaces, where ever they are placed. Try the example below

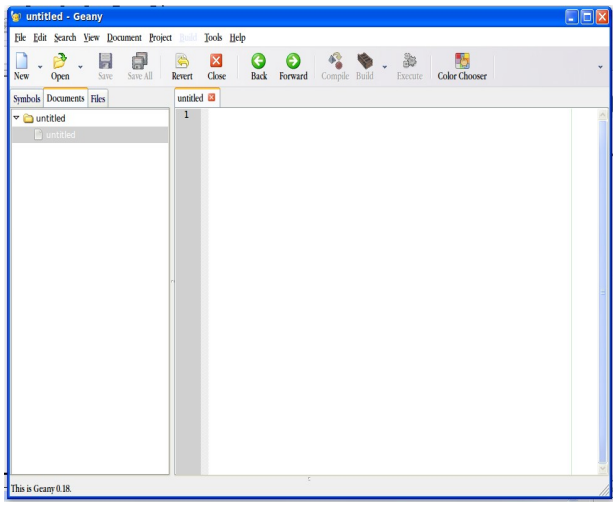
```
>> puts "Tabs \t leave\tlong spaces"
Tabs    leave       long spaces
=> nil
```

I hope you have understood something about Strings, lets move on.....

Using Text Editor

Till now you have keyed in small programs into your irb, when you are developing large software you can't expect the end user or your clients to keep keying in into the console, instead you will be handing over a typed Ruby program which they can run it to accomplish certain task. Lets see how to use a text editor to write programs.

Earlier in Installing IDE section I have typed about how to install a simple Integrated Development Environment (IDE) called Geany. If you are using Ubuntu, goto Applications → Programing → Geany to launch it. You will see a window (well some what similar) as shown.



You can use other IDE's too, if you do refer to their documentation. In the IDE type the following program

```
puts "Hello World!"
puts "This time I used text editor"
```

Now save the file as *hello_world.rb* in a directory, note that Ruby files ends with .rb (dot rb) extension. Launch your terminal / console, migrate to the directory where program is stored and type the following in it

```
ruby hello_world.rb
```

and here's how you will get the output.

```
Hello World!
This time I used text editor
```

Wonderful! You have learned to program with a text editor, you are getting professional aye!

Printing Something

Study the code *hello_world.rb* , we have used a Ruby command called *puts* , this commands puts something to the output, in this case your terminal window.

```
puts "Hello World!"
puts "This time I used text editor"
```

The first line prints *Hello World!* and the second one prints *This time I used a text editor* . What if you want to print two things in the very same line? For it Use the *print* command, lets type a new program *hello_world_1.rb* for it, in your text editor type the following code

```
print "Hello World! "
print "Once again I used a text editor"
```

This gives the output:

```
Hello World! Once again I used a text editor
```

So you have learned to print something!

Getting Input

A program is more useful when it interacts with the user, lets write a program that asks us our name and says hello to us. Type the following code (I saved it as *say_hello.rb*)

```
puts "Hello I am Zigor, a automated Robot that says Hello"
print "Please enter your name:"
name = gets()
puts "Hello #{name}"
```

Now run it, this is how the output will look like

```
Hello I am Zigor, a automated Robot that says Hello
Please enter your name:Karthik
Hello Karthik
```

Lets walkthru the program

The first line

```
puts "Hello I am Zigor, a automated Robot that says Hello"
```

Prints that the program name is Zigor and its a automated robot that wishes you Hello. Then it prints a line feed, hence the content thats printed then on goes to the next line

The the second line

```
print "Please enter your name:"
```

prints out `"Please enter your name:"` , note that we have used `print` here, not `puts` because we want to get the user's name right after `name:`, I feel it will be awkward if we let them type name in the next line, so to avoid the line feed I am using `print` instead of `puts`.

When the user enters name and presses enter, it is caught by the `gets()` function and the thing you typed is stored in the variable called name because of this piece of code

```
name = gets()
```

Now all our Zigor needs to do is to wish hello for which we use this code

```
puts "Hello #{name}"
```

Notice how we are embedding the variable name into string by putting it between `#{` and `}`. The same effect can be achieved by using code like this

```
puts "Hello "+name
```

But doesn't the former piece of code look better? Its all your choice. Ruby let you do the same thing in many ways. You can choose anything that you feel comfortable.

Any way in this topic the line you must be looking at is one that has `gets()` method or function, it waits for a keyboard input, when you give an input and press enter, it takes your input and assigns the value to variable, in this case the variable is name.

Comments

Comments are small pieces of notes you can put into a program so that you or some one else when going through the program 7,658 years from now will remember or come to know what its doing. You may be smart today, but tomorrow you may not be as smart as you are now, your boss or client who has paid you will yell upon you at that moment to fix a priority bug or to update a software. You open your dot rb file and see this code

```
puts "Hello I am Zigor, a automated Robot that says Hello"
print "Please enter your name:"
name = gets()
puts "Hello #{name}"
```

You might be able to understand it now, but after 7,658 years[7]? At that time you might have forgotten Ruby altogether! So start commenting. See the same program *comment.rb* below, how it looks like ?

```
# The client is an idiot
# he wants me to update a software after 7,658 years.
# The hell with him
puts "Hello I am Zigor, a automated Robot that says Hello" # zigor is some ⇨
stupid robot
print "Please enter your name:" # Tells the user to enter his name
name = gets() # gets the user name and assigns it to a variable named name
puts "Hello #{name}" # Embeds name into the string that gets printed
```

Look at the code above, you have told something about client in the first three lines. These lines start with a `#` (hash or check sign). The thing that follows after a check sign is a comment, comments don't interfere with programs execution but it can be used to provide visual hints to humans of whats going on in the program.

Now lets look at this line

```
puts "Hello #{name}" # Embeds name into the string that gets printed
```

here you have `#{name}` enclosed within double quotes, hences its treated as a embedded ruby code in a string rather than a comment, whereas `# Embeds name into the string that gets printed` is treated as comment.

So I hope you understand that comment can one day help. Professionals always comment when they write code. They will take pains that almost any Ruby coder who reads their program will be able to understand how it woks.

Multiline Comments

If you want to put lot of comments, about the size of a paragraph, then you can put that piece of text between `=begin` and `=end` as shown in the program *comments_multiline.rb* below

```
=begin
 The client is an idiot
 he wants me to update a software after 7,658 years.
 The hell with him
=end
puts "Hello I am Zigor, a automated Robot that says Hello" # zigor is some ⇨
stupid robot
print "Please enter your name:" # Tells the user to enter his name
```

7 You can live so long if science progresses fast enough. Researches have data to make you live so long! So be hopeful.

```
name = gets() # gets the user name and assigns it to a variable named name
puts "Hello #{name}" # Embeds name into the string that gets printed
```

In the code above note how we put these text:

```
 The client is an idiot
 he wants me to update a software after 7,658 years.
 The hell with him
```

between *=begin* and *=end* , when you execute the program, those between the *=begin* and *=end* will be ignored. So don't hesitate to write a lot of comment, as now you know there is a way to do it and it will benefit you and your fellow programmers greatly.

Comparison and Logic

Logical Operators

Logical operators lets you determine weather some thing is true or not. For example one is one, thats what humans think, lets see what computers think about it. Fire your irb and type one equals to one as shown

```
>> 1 == 1
=> true
```

Well, whats that double equal to sign? A single equal to sign means assignment, for example a = 5 , puts value 5 into a. A double equal to sign is comparison. So above we have checked if 1 is equal to 1 and the answer is true. Computers are intelligent, aren't they?

OK, now lets check if 1 equals to 2, so we type 1==2 and....

```
>> 1 == 2
=> false
```

The computer (Ruby interpreter in this case) tells its false. Well .. what to say?[8]

Fine, if 1 is not equal to 2 to a computer when we type it, it must putout true, so type it in your console

```
>> 1 != 2
=> true
```

The != stands for not equal to. The ! Stands for not

Now we check if one is not equal to 1 and the computer as expected gives false as output.

```
>> 1 != 1
=> false
```

We now check if 2 is greater than 3, for greater than, we use > sign

```
>> 2 > 3
=> false
```

Oh! 2 is not greater than 3, poor poor 2 :-(

Lets get more intelligent here, we will check if 2 is less than 3, for less than we use < sign

```
>> 2 < 3
=> true
```

Cool! We found that if 2 is not greater than 3, then its less than 3. Well we are going to get a Nobel

8 Read a mathematical proof that computers do go wrong

prize for Math[9].

The >= stands for greater than or equal to

```
>> 5 >= 3
=> true
```

Since 5 is greater than 3, it returns true

See the expression below, it still returns true because 5 is equal to 5

```
>> 5 >= 5
=> true
```

5 is not greater than 5 so it returns false

```
>> 5 > 5
=> false
```

3 is less than 5 hence the less than or equal to operator <= returns true

```
>> 3 <= 5
=> true
```

3 is equal to 3 hence the less than or equal to operator still returns true

```
>> 3 <= 3
=> true
```

3 is not less than 3, its equal to 3 (similar to all humans are equal), hence the less than operator returns false

```
>> 3 < 3
=> false
```

You can also try these with numbers

Operator	Meaning
!<	Not less than
!>	Not greater than

true != “true”

In the logic operator section you might see that irb gives true or false as output. You mustn't confuse with “true” and “false”. The `true` and `false` are logical values whereas “true” and “false” are string.

9 When contacted, unfortunately the Nobel committee told me that if I had discovered 2 is less than 3 in 80,000 B.C it would award me the Nobel prize. Some crack seemed to have discovered it before I did. Any way no problem, if I can invent a time machine and goto 80,000 BC and announce my great discovery I will get the Nobel. So I might on or off writing the book as I have to concentrate my energy on inventing Time Machine.

if

The *if* keyword is used to execute a statement if a condition is satisfied. Take a look at the program below. Execute it.

```
# if.rb

puts "Whats your name?"
name = gets.chop
puts "#{name} is genius" if name == "Zigor"
puts "#{name} is idiot" if name != "Zigor"
```

This is how thee result would be if you give a name other than Zigor

```
Whats your name?
Karthik
Karthik is idiot
```

Take a look at the program. Take a look at the following line

```
puts "#{name} is genius" if name == "Zigor"
```

The program gets your name in variable called *name* . Now it checks if the *name* is Zigor in the code highlighted above, if yes it executes the statement associated with it, in this case it prints out that the particular name is genius. It then comes down to next statement

```
puts "#{name} is idiot" if name != "Zigor"
```

In this statement it checks if *name* is not Zigor, if yes it prints the name is idiot.

if else

Lets write the who's genius program in another form, here we use *if else* condition instead of *if*. Take a look at the code below

```
# Zigor says if the person is intelligent or not
print "Enter your name: "
name = gets.chop
if name == "Zigor"
      puts "#{name} is intelligent"
else
      puts "#{name} is idiot"
end
```

The program when executed gives the same output as previous *if.rb* , whats different is how the logic is represented inside the program. We see a thing called *if name == "Zigor"* , then what has to be executed if the code is true comes after that as shown

```
if name == "Zigor"
      puts "#{name} is intelligent"
```

Now we can put any number of statements after that *if* and all will be executed if the condition given is satisfied. Fine till now, but how will Ruby know where the if statement gets over? To say

that things end here we put an end keyword as shown.

```
if name == "Zigor"
      puts "#{name} is intelligent"
end
```

Lets say that that condition(s) given in the if is not satisfied and we need to do something if condition is invalid, then we put those statements that gets executed when conditions fails under the else keyword as shown

```
if name == "Zigor"
      puts "#{name} is intelligent"
else
      puts "#{name} is idiot"
end
```

Note that the *else* and statements that needs to be executed when condition fails comes before the *end* statement.

elsif

When we use *if* and *else*, the code under *if* gets executed if the condition is satisfied, else the code under *else* section gets executed. Lets have a new scenario where the code under *if* is not satisfied, then the program immediately jumps to the *else* section, now the logic demands that we need to check another condition at the *else* level too, what should we do. To deal with such a scenario we can use the *elsif* command. Take a look at the code below

```
# elsif.rb
# finds the greatest of three numbers
a,b,c = 3,7,5
if a > b and a > c
puts "a = #{a} is greatest"
elsif b > c and b > a
puts "b = #{b} is greatest"
else puts "c = #{c} is greatest"
end
```

When executed it produces the following result

```
b = 7 is greatest
```

Lets walkthru the code step by step. Lets look at the line

```
a,b,c = 3,7,5
```

In this line we assign values 3, 7 and 5 to variables *a,b* and *c*. Lets now come to the if statement

```
if a > b and a > c
```

In this statement we check if *a* is greater than *b* and if *a* is greater than *c*. **Note the keyword *and*.** The *if* condition is satisfied only if both conditions are true. *a* is less than *b* hence this condition fails so program skips the *if* statement and comes to the *elsif* statement

```
elsif b > c and b > a
```

elsif is else plus if, here we check on another two conditions thats separated by and, we check if *b* is greater than *a* and if *b* is greater than *c*, both are true and hence the statement under *elsif*

```
puts "b = #{b} is greatest"
```

gets executed and we get the result. Since the *elsif* is satisfied other *else* and the code that comes under it is ignored.

unless

Unless is another way to check a condition. Let say that one is not a major and is considered a child unless he or she is less than 18 years old. So how to code it in Ruby? Consider the program below, type it in a text editor and execute it.

```
# unless.rb
print "Enter your age:"
age = gets.to_i
p "You are a minor" unless age >= 18
```

When executed this is what we get

```
Enter your age:16
"You are a minor"
```

The program asks your age, it says you are minor if age is not greater than 18. That is it says you are a minor if unless your age is greater than or equal to 18 (see the highlighted code). The p is a kind of short form for puts. If you write *puts "something"* , the ruby interpreter prints *something*. If you use *p "something"* , the ruby intreprete prints *something*.

unless else

Just like *if* with *else*, we can have *else* in an *unless* statement. Type in the program below and execute it

```
# unless_1.rb
print "Enter your age:"
age = gets.to_i
unless age >= 18
      p "You are a minor"
      else p "You are a grown up"
end
```

This is what you get when you execute it

```
Enter your age:37
"You are a grown up"
```

OK, here is how it works , you get your age, convert it into integer and store it in a variable called

age. Concentrate on the highlighted code, unless the age is less than 18 “You are a minor” doesn't get printed out. If the age is greater than or equal to 18 it gets routed to the else statement and “You are a grown up” gets printed out. Note that if we use *else* with *unless* we must terminate the unless block with an *end* command.

Lets now look at another program that uses the unless else. We want to hire people for armed forces, the person should be between 18 and 35 years of age, our program asks the details from a person who wishes to enroll, it checks his age and tells the result. Type the program below and execute it

```
# unless_2.rb
print "Enter your age:"
age = gets.to_i
unless age < 18 or age > 35
      p "You can enter Armed forces"
      else p "You cannot enter Army. You are either too young or too old"
end
```

When executed this will be the result

```
Enter your age:23
"You can enter Armed forces"
```

I think you can explain this program on your own. If else contact me, I will write an explanation.

case when

Suppose you want to write a program that has a determined output for determined input, you can use the case when. Lets say that we want to write a program that spells from 1 to 5, we can do it as shown in *case_when.rb* , type the program in text editor and execute it.

```
# case_when.rb
# This program spells from one to five

print "Enter a number (1-5):"
a = gets.to_i
spell = String.new

case a
      when 1
      spell = "one"
      when 2
      spell = "two"
      when 3
      spell = "three"
      when 4
      spell = "four"
      when 5
      spell = "five"
      else
      spell = nil
```

```
end

puts "The number you entered is "+spell if spell
```

Output

```
Enter a number (1-5):4
The number you entered is four
```

Lets see how the above program works. First the user is prompted to enter a number, when he does enters a number, it gets converted from String to Integer by the following statement

```
a = gets.to_i
```

The variable *a* now contains the value of number we have entered, we have the case statement as shown

```
case a
      ……..
end
```

In the above empty case statement we are going to write code that gets executed depending on the value of *a*. When *a* is 1 we need to spell out as “one” so we add the following code

```
case a
      when 1
      spell = "one"
end
```

Similarly we add code till the case is 5 as shown

```
case a
      when 1
      spell = "one"
      when 2
      spell = "two"
      when 3
      spell = "three"
      when 4
      spell = "four"
      when 5
      spell = "five"
end
```

There could be a case when the human who runs this program could give a wrong input, so we need to deal with those cases too for that we add a special statement called *else*, if all the *when* cases fails, the code under else is executed, it must however be noted that its not mandatory to have an *else* between case … end block. So now the program changes as shown

```
case a
      when 1
      spell = "one"
      when 2
      spell = "two"
      when 3
      spell = "three"
      when 4
```

```
      spell = "four"
      when 5
      spell = "five"
      else
      spell = nil
end
```

Next all we must do is to print out spell which we do it in the following statements

```
puts "The number you entered is "+spell if spell
```

Note that we print out only if *spell* contains a value, else if spell is nil nothing is printed. It is taken care by the if condition thats been highlighted above.

Sometimes it might be necessary that we need to execute same set of statements for many conditions. Lets take a sample application in which the program determines a number from 1 to 10 (both inclusive) and the program tells weather the number is odd or even. Type the code below (*case_odd_even.rb*) and execute it

```
# case_odd_even.rb

num = 7 # put any number from 1 to 10

case num
      when 1, 3, 5, 7, 9
      puts "#{num} is odd"
      when 2, 4, 6, 8, 10
      puts "#{num} is even"
end
```

Output

```
7 is odd
```

Notice that in above program we assign a value 7 to a variable *num*, next we put the *num* in a case statement. When the number is 1, 3, 5, 7 and 9 we need to print its odd so all we do is to group the cases. When its satisfied it must print as odd, for that its just enough if you put it as shown in highlighted code below

```
case num
      when 1, 3, 5, 7, 9
      puts "#{num} is odd"
end
```

Next all we need to print the number is even if its 2, 4, 6, 8 and 10, to do this task all we need to do is to add code that highlighted below

```
case num
      when 1, 3, 5, 7, 9
      puts "#{num} is odd"
      when 2, 4, 6, 8, 10
      puts "#{num} is even"
end
```

Thats it. The code will work fine for all numbers from 1 to 10. The moral of the story is we can

easily group cases and execute a common code under it.

? :

The ?: is called tertiary operator. It can be used as a simple *if*. Take the program shown below. Concentrate on the highlighted code below

```
# max_of_nums.rb

a,b = 3,5
max = a > b  ? a : b
p "max = "+max.to_s
```

When executed the program gives the following output

```
"max = 5"
```

Well the *?:* works as follows. It syntax is like this

<evaluate something > ? <if true take this thing> : <if false take this thing>

You give an expression before the question mark. This expression must either return true or false. If the expression returns true it returns the stuff between *?* and *:* , if false it returns the stuff after :

In the expression

```
max = a > b  ? a : b
```

We can substitute the values of *a* and *b* as follows

```
max = 3 > 5  ? 3 : 5
```

3 is not greater than 5, hence its false. Hence the value after : is assigned to max. Hence max becomes 5.

Loops

At times you might need to do some repetitive task lets say that I want to write a rocket countdown program, I want to create a automated robot that count down for rockets, when the count is finished it says “Blast Off”, lets write one and see

```
# count_down.rb

# Zigor tells about itself
puts "Hello, I am Zigor...."
puts "I count down for rockets"
# Count down starts
puts 10
p 9 # p is a short form for puts
p 8
p 7
p 6
p 5
p 4
p 3
p 2
p 1
p "Blast Off!"
```

Well I hope you understand the program above. There is one thing I would like to explain, p is a short form of puts, rather than writing puts one can use p and get the same result. The above program when run prints the following

```
Hello, I am Zigor....
I count down for rockets
10
9
8
7
6
5
4
3
2
1
"Blast Off!"
```

So a perfect execution, but we can make this more efficient, we will soon see how

downto

In your text editor type the following program

```
# count_down_1.rb
```

```
# Zigor tells about itself
puts "Hello, I am Zigor...."
puts "I count down for rockets"
# Count down starts
10.downto 1 do |num|
      p num
end
p "Blast Off!"
```

Run it and see. Well your program uses up a lot less code and yet it produces the same result! To know how the program runs, look at the code highlighted notice the thing `10.downto 1` , this statement make Zigor count down from 10 to 1 , while it count downs you can do some thing with the countdown value , you can put some code in the loop block. The loop starts with a `do` and ends when it encounters a `end` command. Any code you put should be between the `do` and `end` block[10] as shown below

```
10.downto 1 do
      # do some thing! Anything!!
end
```

So between the do and end (technically its called a block) you can put the code to print the count down number. First how to get the number, we will get it in a variable called num, so we rewrite the code as shown

```
10.downto 1 do |num|
      # put the printing stuff here
end
```

Notice above that num is surrounded by| and |. All we need to do now is to print it, so we just print it!

```
10.downto 1 do |num|
      p num
end
```

times

times is a very simple loop, if you want to get a code executed N number of times you put the code in it. Now lets see what Zigor knows

```
# times.rb

puts "Hi, I am Zigor"
puts "I am going to tell what I know"
7.times{
      puts "I know something"
}
```

Well when executed the program prints the following

10 You can use open and closed flower / curly brackets { and } instead of do and end in Ruby

```
Hi, I am Zigor
I am going to tell what I know
I know something
I know something
I know something
I know something
I know something
I know something
I know something
```

Zigor tells that it knows something seven times.

OK we have made changes in the program, we are printing the count variable this time, type the program below and execute

```
# times_1.rb

puts "Hi, I am Zigor"
puts "I am going to tell what I know"
7.times{ |a|
      puts "#{a}. I know something"
}
```

Here is what you get the result

```
Hi, I am Zigor
I am going to tell what I know
0. I know something
1. I know something
2. I know something
3. I know something
4. I know something
5. I know something
6. I know something
```

Why its counting from zero to six rather than one to seven? Well if all happens as you want, there will be no need of programmers like you and me, so don't bother. Notice that in these programs we use *{* and *}* rather than *do* and *end*. Well, Ruby encourages different styles of programming.

upto

upto counts some number ***upto*** some other number. Its like *downto* in reverse. Type in the program below and execute it

```
# upto.rb

# upto is downto in reverse
17.upto 23 do |i|
      print "#{i}, "
end
```

And here is how the output looks like

```
17, 18, 19, 20, 21, 22, 23,
```

step

step loop can be thought as combination of *upto* and *downto* all packed in one, execute the code shown below

```
# step_1.rb
# explains step function
1.step 10 do |i|
 print "#{i}, "
end
```

and here is the result. This is very similar to upto! Don't you see!!

```
1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
```

Now lets modify the program as shown below and save it in another name

```
# step_2.rb
# explains step function
10.step 1 do |i|
 print "#{i}, "
end
```

When executed this program produces no output. What have we done wrong? Modify the program as shown below and run it

```
# step_3.rb
# explains step function
# this time its stepping down
10.step 1, -1 do |i|
 print "#{i}, "
end
```

Well here is the output of the program

```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1,
```

What goes on in step? step receives three inputs, consider the code shown below

```
10.step 1, -1
```

The first one is the number that calls step is taken as the initial number, in the above case it is 10. Next is the ending number in this case it is 1, that is this function counts from 10 to 1, we must descend in this case, so the count must be in steps of -1.

I can modify the same program to print even numbers in 10 to 1 as shown

```
# step_4.rb
# explains step function
# this time its stepping down
p "Even numbers between 10 and 1:"
10.step 1, -2 do |i|
      print "#{i}, "
end
```

This program prints the following output

```
"Even numbers between 10 and 1:"
```

```
10, 8, 6, 4, 2,
```

Lets now try a program that will print even numbers from 1 to 10, this time in ascending order

```
# step_5.rb
# explains step function
# this time its stepping upby two counts each loop
p "Even numbers between 1 and 10:"
2.step 10, 2 do |i|
      print "#{i}, "
end
```

Output

```
"Even numbers between 1 and 10:"
2, 4, 6, 8, 10,
```

See the highlighted or darkened code above. We have started from 2,we will end at 10 and we jump each loop by steps of 2. Inside the loop we simply print the iterating value which is captured in variable `i`.

while

While loop is a loop that does something till a condition is satisfied. Read the code below

```
# while.rb
i=1
while i<=10 do
      print "#{i}, "
      i+=1
end
```

when executed, it produces the following output.

```
1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
```

Lets now see how an while loop works. A while loop normally has four important parts

5. Initialization
6. Condition check
7. Loop body
8. Updation

Initialization

See the statement `i=1` , here we initialize a variable named `i` and set it to value 1.

Condition check

See the statement `while i<=10` , in this statement we specify that we are starting a while loop, this

while loop on every iteration checks the value of `i`,if its less than or equal to 10 , the loops body gets blindly executed

Loop body

Notice the `do` and `end` in the program. They encapsulate a piece of code. The `do` symbolizes the start of loop code block, the `end` symbolizes the end of loop code block. Between it we have some statements about which we will discuss soon. One of the statement is to print the value of `i`, which is accomplished by `print "#{i}, "`

Updation

Lets say that we forgot to include `i+=1` in the loop body, at the end of each iteration the value of `i` will always remain 1 and `i` will always remain less than 10 hence the loop will be executed infinite number of times and will print infinite 1's, . In practical terms your program will crash with possible undesirable consequence. To avoid this we must include a updation statement. Here we have put `i+=1` which increments `i` by value one, every time an iteration continues, this ensures that `i<=10` to become false at some stage and hence the loops stops execution[11].

Hence we see that for a loop to work in an desirable manner we need to get these four parts into symphony.

until

`While` loop keeps going until a condition becomes false, `until` loop keeps going until a condition becomes true. Read the code below, type it in a text editor and execute it.

```
# until.rb
i=1
until i>10 do
      print "#{i}, "
      i+=1
end
```

This is what you will get as result

```
1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
```

So how this loop works? At first we do set `i=1` , then we use the until command and say that until `i` is greater than 10 keep doing some thing (look at the highlighted code). What should be done is said between the `do` and `end` key words. So till the condition fails the code in loops body will be

11 Some cases a loop might be let to run infinite times (theoretically). Currently those things are outside the scope of this book.

executed, so we get 1 to 10 printed as output.

break

Suppose you want to break away from loop, you can use the *break* command. An example is given below. In the example we will break if the iterating variable *i* becomes 6. So numbers ranging only from 1 to 5 gets printed. When *i* becomes 6 the loop breaks or terminates

```
#break.rb

1.upto 10 do |i|
      break if i == 6
      print "#{i}, "
end
```

When executed, the above program produces the following output

```
1, 2, 3, 4, 5,
```

Arrays

Arrays can be considered as rack, you can keep any thing[12] in a rack, in a similar way you can keep any thing in an array. A rack contains many shelfs or compartments. If you can count them you can put numbers on each compartment, the rack can be considered an array of space to store some thing. Each compartment can be identified by a number and hence it becomes easy to identify it. An array is a rack thats available to a programmer. Lets see an example to learn more. Type the program below and execute it

```
# array.rb

my_array = []
my_array << "Something"
my_array << 123
my_array << Time.now

my_array.each do |element|
puts element
end
```

This is how you will get the output

```
Something
123
Tue Feb 02 18:10:06 +0530 2010
```

Lets walkthru the program, take the line `my_array = []` , in it we declare an array called `my_array`, its an empty array that has got nothing in it. `[]` denotes an empty array and we assign my_array to it. Having done so we populate it with some values. In the following statements

```
my_array << "Something"
my_array << 123
my_array << Time.now
```

We append elements to an array. In the first statement we append a string constant `"Something"`, in the second statement we append a integer `123` and in the third statement we append the current time. If you have guessed it right, we have used `<<` operator to append the array.

Till now we have created an array called `my_array` and have put something into it. Now we have to see what we have put in. To do so we use `<array_name>.each` (array name dot each). This method extracts each element of an array. So for `my_array` we use

```
my_array.each
```

12 Things are called objects and classes in programming

OK we have to do some thing with each element of an array. To do so we add a *do … end*, within it we can do something, so our code gets transformed as

```
my_array.each do
end
```

We have to capture each element of an array into a variable, lets use a variable named *element* to do the job, so we capture each element using the following code

```
my_array.each do |element|

end
```

Notice that how we put our element variable between | and |. We have captured each and every element of an array, what to do? we will print it using a puts statement. So our array gets printed successfully. The following program too works the same way as previous program, but we use *Array.new* instead of *[]* to say that *my_array* is an array

```
# array_1.rb

my_array = Array.new
my_array << "Something"
my_array << 123
my_array << Time.now

my_array.each do |element|
puts element
end
```

I will write another program that will use the *for* construct to iterate over each element of an array as shown below

```
# array_2.rb

my_array = Array.new
my_array << "Something"
my_array << 123
my_array << Time.now

for element in my_array
      puts element
end
```

Output

```
Something
123
2012-08-10 19:19:47 +0530
```

There is a third way of creating an array. Take a close look at the program below. Look at the darkened statement. Look at it carefully, we have my_array thats an array variable its been set to *["Something", 123, Time.now]*. That is in this case we declare and add array elements to my_array in the same statement. Note that we put elements of an array in square brackets, this is

another way of declaring array. So the program *array_3.rb* works exactly same as *array_1.rb* and *array.rb*, but its more concise. Unlike many languages, Ruby lets the programmer choose his own style of coding.

```
# array_3.rb

my_array = [ "Something", 123, Time.now ]
puts my_array.join("\n")
```

Result

```
Something
123
Wed Feb 03 17:37:36 +0530 2010
```

More on Array

Lets now see some array functions. For this we will be using our favorite irb rather than a text editor

```
>> array = Array.new
=> []
```

OK in the above statement we see that we create an Array named array using Array.new . Array.new creates an empty array.

There is another way to create an array. We can create it by directly specifying the values that are contained in an array as shown

```
>> array = ["Something", 123, Time.now]
=> ["Something", 123, Tue Feb 02 20:30:41 +0530 2010]
```

In the above statement, we create an array with three objects in it. The value that must be in an array is given between square brackets [and]. Each object in array is separated by a comma. By providing no values between [and] we can even create an empty array as shown

```
>> array = []
=> []
```

In the above example the empty [] does the same job as Array.new .

Lets create array with parameters with Array.new as shown

```
>> array = Array.new("Something", 123, Time.now)
ArgumentError: wrong number of arguments (3 for 2)
      from (irb):3:in `initialize'
      from (irb):3:in `new'
      from (irb):3
      from :0
```

As you see above it fails! Don't use it that way.

OK, lets now try some thing on the array, first to get how many elements are in the array we can use

the length function as shown below:

```
>> array.length
=> 3
```

The join function joins many array elements together and returns it. So when our array element is joined this is what we get as result:

```
>> array.join(', ')
=> "Something, 123, Tue Feb 02 20:30:41 +0530 2010"
```

Note that we pass a string ', ' to the join, when the array elements are joined as a string the, string we passed gets inserted into them in between.

We have created an array and we have something in it, what if we want to add something to it? To do so we use the push method. In the example below, we push a number 5 into the array and as we see the array gets expanded and 5 is appended to the array at the last.

```
>> array.push(5)
=> ["Something", 123, Tue Feb 02 20:30:41 +0530 2010, 5]
```

The pop method does the opposite of push, it pops out or removes the last element of array. See the example below, we pop an element and the last element which is 5 gets popped out.

```
>> array.pop
=> 5
```

After popping it out lets see whats in the array

```
>> array
=> ["Something", 123, Tue Feb 02 20:30:41 +0530 2010]
```

We see that the array size has reduced by one and last element 5 is missing.

Its not that you must only give a fixed values in push, you can give variables and Ruby expressions and any object to the push as argument. You can see below that we are pushing 2 raised to the power of 10 to the array and 1024 gets added to the array at the last.

```
>> array.push 2**10
=> ["Something", 123, Tue Feb 02 20:30:41 +0530 2010, 1024]
```

Array elements are indexed. The first element of an array has a index number 0 and its goes on (theoretically till infinity). If one wants to access element at index n[13], all he needs to do is to put the index number in between square brackets. In the example below we access the third element in the array named array so we type it as follows

```
>> array[2]
=> Tue Feb 02 20:30:41 +0530 2010
```

The pop method too accepts a Fixnum[14] as an argument which it uses to pop all elements starting

13 n is a number
14 A number

from that index and further.

```
>> array.pop(2)
=> [Tue Feb 02 20:30:41 +0530 2010, 1024]
>> array
=> ["Something", 123]
```

As you see the third element gets popped out, so popping at random is possible.

We can push many elements into an array at once. Consider the code snippet below

```
>> array.push 5, "Who am I?", 23.465*24
=> ["Something", 123, 5, "Who am I?", 563.16]
```

We first push 3 new elements into the array and so we get a bigger one.

Now we pop all elements who's index number is 3 and above by giving *array.pop 3*

```
>> array.pop 3
=> [5, "Who am I?", 563.16]
```

As you can see the array size is reduced and it now only has two elements.

```
>> array
=> ["Something", 123]
```

There is another way to append elements in an array, its by using the double less than operator <<, let push some elements into the array with it as shown:

```
>> array << "a new element"
=> ["Something", 123, "a new element"]
>> array << 64
=> ["Something", 123, "a new element", 64]
```

as you see above we have appended a String constant *"a new element"* and *64* to the array using *<<* operator.

You can find maximum and minimum values in an array using the max and min function as shown:

```
>> nums = [1, 2, 64, -17, 5 ,81]
=> [1, 2, 64, -17, 5, 81]
>> nums.max
=> 81
>> nums.min
=> -17
```

As you see in above example we create a array called *nums* having some numbers, *nums.max* returns the maximum value in that array and *nums.min* returns the minimum value in that array.

Set operations

For those who know set theory must know about intersections, unions and blah blah. I read about set theory when in school and now have forgotten about it. You can treat array as set and do many operations on it. Here are a few examples which I tried out on irb

Lets take a college volleyball team, in it are some people names Ashok, Chavan, Karthik, Jesus and Budha. If you take a list of cricket team there are Budha, Karthik, Ragu and Ram. Lets now code it in ruby. To have a collection of people who play in volleyball team we create an array as shown

```
>> volleyball=["Ashok", "Chavan", "Karthik", "Jesus", "Budha"]
=> ["Ashok", "Chavan", "Karthik", "Jesus", "Budha"]
```

In a similar way we create another array that contains names of those who play in cricket team as shown

```
>> cricket=["Budha", "Karthik", "Ragu", "Ram"]
=> ["Budha", "Karthik", "Ragu", "Ram"]
```

So we have two sets of people. Now to find out who are in volley ball and cricket, all we need to do is to AND (or take intersection of) both arrays using the `&` operator as shown

```
>> volleyball & cricket
=> ["Karthik", "Budha"]
```

As you see from above code snippet, the `&` (and) operator sniffs out those elements that are there in both arrays. In mathematics this stuff is called intersection.

Lets say in another situation we would like to find out all those who are both in volleyball and cricket team. To do so we use the or operator | . Lets now apply it

```
>> volleyball | cricket
=> ["Ashok", "Chavan", "Karthik", "Jesus", "Budha", "Ragu", "Ram"]
```

As you see we get names of those who are in volleyball and cricket team. The | (or) operator is different from the + (plus) operator. Lets add volleyball and cricket teams

```
>> volleyball + cricket
=> ["Ashok", "Chavan", "Karthik", "Jesus", "Budha", "Budha", "Karthik", "Ragu",
"Ram"]
```

As you can see from above code snippet the names Karthik and Budha are duplicated. This does not happen when we use the | (OR) operator.

Lets now find that which players play only for the volleyball team. For this we will minus the players of cricket from the volleyball team using the – (minus) operator as shown

```
>> volleyball - cricket
=> ["Ashok", "Chavan", "Jesus"]
```

So we see three players are exclusively in volleyball team. So if you are a mathematician you will feel some what comfortable with Ruby.

Hashes and Symbols

Hashes are arrays with index defined by the program or user and not by the Ruby interpreter. Lets see a program to find out how hashes work. Type the following program (*hash.rb*) into your text editor and execute it.

```
#!/usr/bin/ruby
# hash.rb
mark = Hash.new
mark['English'] = 50
mark['Math'] = 70
mark['Science'] = 75
print "Enter subject name:"
sub = gets.chop
puts "Mark in #{sub} is #{mark[sub]}"
```

Output 1

```
Enter subject name:Math
Mark in Math is 70
```

Output 2

```
Enter subject name:French
Mark in French is
```

Take a look at output 1. The program asks the user to enter subject name. When the user enters math, the program gives the math mark. Lets walkthru the code. At the very beginning we have the line *mark = Hash.new* , in this line we declare a variable called mark of the type hash. Take a look at the following lines

```
mark['English'] = 50
mark['Math'] = 70
mark['Science'] = 75
```

Unlike an array, hash can have a object as index. In this case we have used simple string as index. We put the marks obtained in English, Math and Science in *mark['English']* , *mark['Math']* , *mark['Science']*. The next two statements

```
print "Enter subject name:"
sub = gets.chop
```

prompts the user to enter mark, when he does it, it gets stored into the variable called *sub*. In the final line *puts "Mark in #{sub} is #{mark[sub]}"* we simply access the hash value using *sub* as the key and print it out.

Take a look at output 2, in this case I have entered *French* and the program gives out no result. In

the following program we will learn how to deal with it.

Default values in Hash

When we pass the index to an hash and if its value does exist, then the hash will faithfully return that value. What if the index has no value defined. In the previous example we saw the program returns nothing. In this program we hope to fix it. Look at the highlighted or darkened code. In it instead of just giving *mark = Hash.new* as in the previous one, we have given *mark = Hash.new 0* , here the zero is the default value. Now lets run the program and see what happens.

```
#!/usr/bin/ruby
# hash_default_value.rb
mark = Hash.new 0 # We specify default value of mark is zero
mark['English'] = 50
mark['Math'] = 70
mark['Science'] = 75
print "Enter subject name:"
sub = gets.chop
puts "Mark in #{sub} is #{mark[sub]}"
```

Output

```
Enter subject name:Chemistry
Mark in Chemistry is 0
```

Look at the output, we haven't defined a value for *mark['Chemistry']*, yet when the subject name is specified as Chemistry we get *0* as result. This is so because we have set zero as the default value. So by setting default value we will have a value for those indexes we haven't defined yet.

Looping hashes

Looping in arrays is quiet easy, we normally use *each* function in array to iterate objects in array. In similar fashion we can loop in hashes. Type the following code hash_looping.rb into a text editor and execute it.

```
#!/usr/bin/ruby
# hash_looping.rb
mark = Hash.new 0 # We specify default value of mark is zero
mark['English'] = 50
mark['Math'] = 70
mark['Science'] = 75
total = 0
mark.each { |key,value|
      total += value
}
puts "Total marks = "+total.to_s
```

Output

```
Total marks = 195
```

In the program above we have calculated the total of all marks stored in the Hash mark. Note how we use the each loop. Note that we get the key value pair by using |*key, value*| in the loop body. The key holds the index of the hash and value holds the value stored at that particular index[15]. Each time the loop is executed, we add value to total, so at the end the variable total has got the total of the values stored in the hash. At last we print out the total.

Below is another program where we store student marks in an hash, we use the each loop to print the key and value corresponding to the key. Hope you have understood enough to explain the code below all by your self.

```
#!/usr/bin/ruby
# hash_looping_1.rb
mark = Hash.new 0 # We specify default value of mark is zero
mark['English'] = 50
mark['Math'] = 70
mark['Science'] = 75
puts "Key => Value"
mark.each { |a,b|
      puts "#{a} => #{b}"
}
```

Output

```
Key => Value
Science => 75
English => 50
Math => 70
```

More way of hash creation

There is another of hash creation, lets look at it. Shown below, the explanation of the is same like the of previous program *hash_looping_1.rb* , except for the highlighted line in program below. Explain the program to yourself as I don't have the mood to write now

```
#!/usr/bin/ruby
# hash_creation_1.rb
marks = { 'English' => 50, 'Math' => 70, 'Science' => 75 }
puts "Key => Value"
marks.each { |a,b|
      puts "#{a} => #{b}"
}
```

Output

```
Key => Value
Science => 75
English => 50
Math => 70
```

15 The term index and key refer to the same stuff

Using symbols

Usually in a hash we use Symbols as keys instead of String. This is because Symbol occupies far less amount of space compared to String. The difference in speed and space requirement may not be evident to you now , but if you are writing a program that creates thousands of hashes it may take a toll. So try to use symbols instead of String.

So what is symbol? Lets fire up our irb by typing `irb --simple-prompt` in terminal. In it type the following

```
>> :x.class
=> Symbol
```

Notice that we have placed a colon before x thus making it `:x`. Symbols have a colon at their start. When we ask what class is `:x`, it says its a symbol. A symbol is a thing or object that can be used as a key in a hash[16]. In similar way we declare another symbol called name and see what class it belongs.

```
>> :name
=> :name
>> :name.class
=> Symbol
```

A variable can hold a symbol in it. Notice below that we have assigned variable a with value `:apple` which is nothing but a symbol. When we ask what class is a by using `a.class` , it says its a symbol.

```
>> a = :apple
=> :apple
>> a.class
=> Symbol
```

Symbols can be converted to string using the `to_s` method / function. Look at the irb example below where we convert symbol to string.

```
>> :orange.to_s
=> "orange"
```

There is no method in String to convert it to symbol. Suppose we would like to convert a string to symbol, we could do it as shown. In the example below we try to convert a string called “human” to symbol. To do so we employ the following trick

```
>> human = ":"+"human"
=> ":human"
```

First we append colon to a “human” to make it “:human” which still is a String.

```
>> human.class
=> String
```

16 Well it can be used for many things other than that. For now remembering this is sufficient

as we can see above the variable *human* that contains *":human"* is saying that its type is string. What comes next is a cool thing, look at the code snippet below. We have a variable called *human_sym* to it is assigned *eval(human)*. What that eval? *eval* is a function in Ruby that evaluates any String passed to it as a Ruby program! This is a really powerful feature. Substituting the value of *human*, it becomes *eval(":human")* and *eval* evaluates it as a symbol which gets stored in variable *human_sym*.

```
>> human_sym = eval(human)
=> :human
>> human_sym.class
=> Symbol
```

So when we call the class method for *human_sym*, it says its a symbol. Thanks to very powerful function *eval*. Having equipped us with this knowledge, let us write a program in which hashes dont use not String but Symbols as key. Type in the program (below) *hash_symbol.rb* into text editor and execute it.

```
#!/usr/bin/ruby
# hash_symbol.rb

mark = Hash.new 0 # We specify default value of mark is zero
mark[:English] = 50
mark[:Math] = 70
mark[:Science] = 75
print "Enter subject name:"
sub = gets.chop
symbol = eval ":"+sub
puts "Mark in #{sub} is #{mark[symbol]}"
```

Output

```
Enter subject name:Math
Mark in Math is 70
```

When the program is run, it prompts for subject name, when its entered it shows corresponding mark. Lets walkthru the code and see how it works. Notice that we use Symbols and not Strings as keys in mark Hash as shown

```
mark[:English] = 50
mark[:Math] = 70
mark[:Science] = 75
```

Next we prompt the user to enter marks by using *print "Enter subject name:"* , the user enters the subject name . Now look at the next three lines, first we get the subject name into variable sub using the following statement

```
sub = gets.chop
```

Having got the subject name we need to convert it into Symbol which is done by using *eval* method as shown below

```
symbol = eval ":"+sub
```

In the statement above colon is concatenated before the value in *sub*, and this is passed to *eval* method which returns a Symbol. This Symbol is stored in variable called *symbol*. So if we have entered subject name as *Math*, the *eval* would have returned *:Math* which gets stored in symbol.

All we do next is to access the array value at key *symbol* and print it using the following statement

```
puts "Mark in #{sub} is #{mark[symbol]}"
```

Well why did Ruby make it so easy to convert a symbol to String and make it so hard to convert a string to Symbol. Well try this in irb

```
>> "hello".to_sym
=> :hello
```

Now rewrite the program above all by your self by using newly gained knowledge.

Ranges

Some times we need to have a range of values, for example in a grading system. If a student scores from 60 to 100 marks, his grade is A, from 50 to 59 his grade is B and so on. When ever we need to deal with a Range of values we can use ranges in Ruby. Type `irb --simple-prompt` in your terminal and type these into it

```
>>(1..5).each {|a| print "#{a}, " }
1, 2, 3, 4, 5, => 1..5
```

OK whats that `(1..5)` in the above statement, this is called Range. Range is a object that has got an upper value and a lower value and all values in between. Note that like array each and every value in a range can be got out using a each method as shown above.

Range does not work only on numbers it can work on strings too as shown below

```
>> ("bad".."bag").each {|a| print "#{a}, " }
bad, bae, baf, bag, => "bad".."bag"
```

Lets try out another few examples in our irb that will tell to us more about Ranges. So fire up your irb and type the following

```
>> a = -4..10
=> -4..10
```

In the above code snippet we create a range that ranges from value -4 to 10. To check what variable type `a` belongs lets find out what class it is

```
>> a.class
=> Range
```

As we can see `a` belongs to range class

To get the maximum value in a range use the `max` method as shown

```
>> a.max
=> 10
```

To get the minimum in a range use the `min` method as shown

```
>> a.min
=> -4
```

Its possible to convert range to an array by using `to_a` method as shown

```
>> a.to_a
=> [-4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Ranges used in case .. when

Look at the program below (*ranges_case_when.rb*), we are building a student grading system in which when a mark is entered the program puts out the grade of the student, study the code, type it and execute it, we will soon see how it works.

```
#!/usr/bin/ruby
# ranges_case_when.rb

puts "Student grading system"
print "Enter student mark: "
mark = gets.chop.to_i

grade = case mark
      when 80..100      : 'A'
      when 60..79       : 'B'
      when 40..59       : 'C'
      when 0..39        : 'D'
      else "Unable to determine grade. Try again."
end

puts "Your grade is #{grade}"
```

Output

```
Enter student mark: 72
Your grade is B
```

If you execute this program in Ruby 1.9.3, it will throw out an error. For Ruby 1.9, this is how you need to code the above program

```
#!/usr/bin/ruby
# ranges_case_when_19.rb

puts "Student grading system"
print "Enter student mark: "
mark = gets.chop.to_i

grade = case mark
      when 80..100
            'A'
      when 60..79
            'B'
      when 40..59
            'C'
      when 0..39
            'D'
      else
            "Unable to determine grade. Try again."
end

puts "Your grade is #{grade}"
```

At first the program prints that the software is student grading system and asks the user to enter the student mark. When the mark is entered its got using *gets* statement, the trailing newline character is chopped using the *chop* method and its converted to integer using the *to_i* method and the mark

is stored in the variable mark. All of it is done using this *mark = gets.chop.to_i* statement.

Once we have the mark we need to compare it with a range of values to determine the grade which is done using the following statements:

```
grade = case mark
      when 80..100      : 'A'
      when 60..79       : 'B'
      when 40..59       : 'C'
      when 0..39        : 'D'
      else "Unable to determine grade. Try again."
end
```

Here we see that mark is passed to the case statement. In the *when*(s) we don't have a number or String to compare the mark, in fact we have ranges. When the mark lies from 80 to 100 (both inclusive) the grade is set to A, when its lies in 60 to 79 its set to B, C for 40 to 59 and D for 0 to 39. If the user enters something wrong, grade will be set to "Unable to determine grade. Try again.".

So as we can see ranges come very handy when they are used with *case when* statement. It makes programming relatively simple when compared to other languages.

Checking Intervals

Another use of Ranges is to check if any thing is located in a particular interval. Consider the program (*ranges_cap_or_small.rb*) below

```
#!/usr/bin/ruby
# ranges_cap_or_small.rb

print "Enter any letter: "
letter = gets.chop

puts "You have entered a lower case letter" if  ('a'..'z') === letter
puts "You have entered a upper case letter" if  ('A'..'Z') === letter
```

Output

```
Enter any letter: R
You have entered a upper case letter
```

Read it carefully and execute it. In the above case I have entered capital R and hence the program says I have entered a upper case letter. If I had entered a lower case letter, the program would have said I had entered a lower case letter. Lets see how the program works. The following lines:

```
print "Enter any letter: "
letter = gets.chop
```

prompts the user to enter a letter, when the user enters a letter the *gets* method gets it, *chop* chops off the new line character thats added due to the enter key we press. In the next line look at the *if*

`('a'..'z') === letter` , here we check if the value in variable letter lies with 'a' and 'z' (both inclusive) , if it does, we print that user has entered small letter.. Note that we don't use double equal to == but we use triple equal to ===[17] to check if its in range. In a similar way `('A'..'Z') === letter` returns `true` if `letter` has capital letter in it and the program prints the user has entered a capital letter.

Using triple dots …

Another thing in Range I would like to add is using triple dots instead of using double dots. Just try these out on your irb.

```
>> (1..5).to_a
=> [1, 2, 3, 4, 5]
>> (1...5).to_a
=> [1, 2, 3, 4]
```

See from above code snippet when I type `(1..5).to_a` we get an array output as `[1, 2, 3, 4, 5]` , but for `(1...5).to_a` we get output as `[1, 2, 3, 4]` . When we use the triple dot the last thing in Range that would appear when we use double dots gets cut out.

Functions

When you are using same piece of code many times, you can group them into a thing called function, you can call that grouped code any where in your program to do that particular task. Lets take a real world example, you goto the hotel and a waiter comes up to you. You order an fish fry and you get it. You are not bothered what happens after you order.

Once you have ordered the fry, there are lots of procedures that takes place. The waiter notes down your order, he goes up to the kitchen and places the order chit to the chef, the chef tells him that it would take so much time for the dish to get cooked. The waiter thinks how he could keep you from getting mad and arrives at your table, recommends a starter and/or an appetizer, serves you a drink that would go well before you eat the dish, he pools the kitchen to see if the dish is ready. If the dish is ready and if you have finished your starter he serves it. If you haven't finished your starter, he

17 This triple equal to = = = is technically called case equality operator. Who who cares?

tells the kitchen to keep the dish warm and waits for you to finish. Once he gets the dish to your table, he lays the plate containing the dish and cutlery.

All you have done is to order a fish fry, and have blissfully ignored what is being *functioned* at the background. You gave some input to a waiter and got a dish (output). What to do and not to is preprogrammed or trained into the waiters mind, according to his training, the waiter functions.

Lets get started with functions in Ruby. We will be looking at a program in which we will be writing a function called *print_line* , which prints a line. Type the following program into your text editor and run it.

```
# function.rb

def print_line
      puts '_'*20
end

print_line
puts "This program prints lines"
print_line
```

This is what you will get as output.

```
____________________
This program prints lines
____________________
```

Lets analyze the program. Consider the following piece of code

```
def print_line
      puts '_'*20
end
```

The *def* tell thats you are defining a function. A function must have a name, the name follows immediately after *def* key word. In this case the name of the function is *print_line*. In it you can write what ever code you want. In this case we are creating a line with twenty underscore characters.

So we have created a function and have added code into it. All we need to do now is to call the function from our program. Its done by just typing the name of the function in the program as highlighted in code below

```
# function.rb

def print_line
      puts '_'*20
end

print_line
puts "This program prints lines"
print_line
```

As seen in the output, a line made up of twenty underscore characters gets printed above and below the string “This program prints lines”.

Argument Passing

Lets get more control of the functions in this section. You don't goto a hotel and order a single dish, you can order how much ever or how less you want. If you go with friends you can order more servings, if you are alone, you will order less. Why can we do the same thing with *print_line* function? Why can't we vary its length, doing so will be a wonderful thing, we can print lines of length what ever we choose.

Take a look at code highlighted below, we have typed a thing called length after the function name, its called as argument. Like a function, an argument has an name, in this case we have named it *length*. We can pass any values to it to vary the length of the line printed. Type the code below and execute it

```
# function_1.rb

def print_line length
      puts '_'*length
end

10.step(50,10) do |x|
      print_line x
end

40.step(10,-10) do |x|
      print_line x
 end
```

You will get a design pattern as shown below

```
__________
____________________
______________________________
__________________________________________
______________________________________________________________
__________________________________________
______________________________
____________________
__________
```

Take a look at the following code

```
10.step(50,10) do |x|
      print_line x
end

40.step(10,-10) do |x|
      print_line x
 end
```

We have used step functions to increment or decrement the value which we capture it into a variable called *x* which is inside the loop, we pass *x* to the *print_line* function by placing it after its call as highlighted above. So each time a line of varying length (determined by *x*) gets printed. The program is constructed in a way so that a pattern is generated.

Default Argument

In *function_1.rb* you have seen how to pass an argument to a function. What if suppose you fail to pass an argument to the function. If you do so, an error will be generated which a good programmer will not desire[18] to happen. To prevent this and to make programming a bit easy its better to provide a default argument to a function. Note the highlighted code given below in function_default_argument.rb

```
# function_default_argument.rb

def print_line length = 20
      puts '_'*length
end

print_line
print_line 40
```

Execute the program and observe the result

You can see in the program, in function *print_line* by giving *length = 20* we have indicated that if no argument is passed the function must assume that value of *length* is 20 . If passed this value will be overridden with what ever value you pass. As you can see in the second highlight (where the function is highlighted), we simply call the function just by its name (*print_line*). We don't bother to pass value for length to it, yet we see a line of , length 20 units gets printed in the output. I hope you know why.

Returning Values

We have till now seen function taking in arguments, we will now see that the function can return back values which can be used for some purpose. Lets now see the program function_return.rb, study the code, type it and execute it.

```
#! /usr/bin/ruby
# function_return.rb
```

18 “Desire is the cause of all suffering” - Buddha

```
def addition x, y
      sum = x+y
      return sum
end

a, b = 3, 5

puts addition a,b
```

Output

```
8
```

The output you get out after executing is 8, which proves that we have written a flawless program. Note the method named *addition* in the above program. It accepts two arguments *x* and *y*, inside the method we declare a variable called *sum* which is assigned to the addition of *x* with *y*. The next statement is the hero here, see that we have used a keyword *return*, this returns the value out of the function. In the above program we return out the *sum* and hence when we get the answer out.

Its not that we must use return statement to return a value. The last statement that gets executed in a Ruby function gets returned by default. Consider the program *function_last_gets_returned.rb* thats shown below. In it we have a method called *square_it* which accepts a single argument called *x*. It has a single statement *x**2* which happens to be the last statement as well.

```
#!/usr/bin/ruby
# function_last_gets_returned.rb

def square_it x
      x**2
end

puts square_it 5
```

Type the program and execute it.

```
25
```

As you see we have called *square_it 5* and we get 25 as the result. Its possible because in Ruby the result of last executed statement gets returned by default.

Recursive function

Lets see another math thing. You might be wondering why am I doing all math? Certain programmers do write books that keeps out as much math out as possible, I am not a professional mathematician, but I admire math. Computers are based on math. All computers use a thing called boolean algebra to do all tasks. I wouldn't say that you must be a mathematician to be a programmer, but knowing math does help.

OK what is a factorial? Take a number, lets take 3, now what will be 3 X 2 X 1 , that will be six!

Simple isn't it? 6 is factorial of 3. Well we will take 4 now, so 4 X 3 X 2 X 1 will be 24, in similar way factorial of 2 will be 2 X 1 which is 2. Having equipped with this knowledge we will now construct a program that will give us factorial of a number.

Study the program given below. Concentrate on the function named factorial

```
# factorial.rb

def factorial num
      fact = 1
      1.upto(num) { |a|
            fact = fact * a
      }
      fact
end

number = 17
puts "Factorial of #{number} = #{factorial number}"
```

Execute the code above and this is what you get as output

```
Factorial of 17 = 355687428096000
```

In the above example (in the function factorial) we have taken all number from one to the particular number, multiplied it and got factorial. Now study the code factorial_1.rb shown below

```
# factorial_1.rb

def factorial num
      return 1 if num == 1
      return num * factorial(num-1)
end

number = 17
puts "Factorial of #{number} = #{factorial number}"
```

Execute the code above and this is what you get as output

```
Factorial of 17 = 355687428096000
```

The output is same as the previous program *factorial.rb* . Take a very close look at the function named factorial in above program. Let me list it out for you to see

```
def factorial num
      return 1 if num == 1
      return num * factorial(num-1)
end
```

This function is confusing, isn't it? When I was crazy and did want to learn about programming I studied C. The concept of recursive functions was explained using factorial, and I never did understand it for a long time. To avoid the pain let me explain in detail.

Take the number 1. Factorial of it is 1. So if 1 is encountered 1 is returned as shown in highlighted code below

```
def factorial num
      return 1 if num == 1
      return num * factorial(num-1)
end
```

Now take the number 2. Factorial of it 2 X 1 , which is 2 multiplied factorial of 1. In other words we can write it as 2 multiplied by factorial of 2-1 ($2 x\ factorial(2-1)$). So if number two is encountered in the function factorial, it skips the first if statement and the second statement thats highlighted below gets executed

```
def factorial num
      return 1 if num == 1
      return num * factorial(num-1)
end
```

In this an interesting thing happens. Here factorial (2-1) is called, that is factorial function calls itself. So when factorial of 2-1 , i.e factorial of 1 is called, it returns 1. 1 is multiplied by 2 and is returned back.

Now take the number 3. Its factorial is 3 X 2 X 1. This can be written as 3 multiplied by factorial 2. Factorial 2 gets translated as 2 multiplied by factorial 1. Hence the result is got out finally. For any number larger than 1, the factorial function calls itself repeatedly. The process of function calling itself is called recursion.

Variable Scope

We have seen about functions in the last section and we have seen about variable before. I think the time is right to type about variable scope. In this chapter we examine how long or how far a variable is valuable when its declared in a particular section of a program. Lets start with a example. Fire up your text editor, type the code below (*variable_scope.rb*) and execute it.

```
#!/usr/bin/ruby
# variable_scope.rb

x = 5

def print_x
      puts x
end

print_x
```

Output

```
variable_scope.rb:7:in `print_x': undefined local variable or method `x' for
main:Object (NameError)
      from variable_scope.rb:10
```

Well you get an error. See that you have declared a variable by typing *x = 5*. In the function *print_x* you tell the ruby program to print out the variable *x*, but it throws a error. Look at the output it says *undefined local variable or method `x' for main:Object (NameError) from variable_scope.rb:10* , well we have defined *x* and have assigned it to value *5* at the beginning, then how come Ruby throws the error. Well we have defined x outside the function *print_x* hence *x* has no scope inside it, so we get an error.

A good programmer is the one who exploits the advantages provided by a programing language and who is smart enough to play by rules and limitations it imposes. It might look as a real handicap to a newbie that we are not able to access a variable we have assigned outside a function, but as you program and become mature programmer you will realize its blessing in disguise.

To learn more type the program below (variable_scope_1.rb) in your text editor and execute it.

```
#!/usr/bin/ruby
# variable_scope_1.rb

x = 5

def print_x
```

```
      x=3
      puts x
end

print_x
puts x
```

Output

```
3
5
```

Take a careful look at the output. First we declare a variable `x = 5` , then in function `print_x` we declare a variable `x = 3`. Note that the variable declared in function `print_x` is not the same one as thats been declared outside the function. Next we call upon the function `print_x` which prints the output as 3 which is expected since inside `print_x` we have written `puts x` after `x =3`. Next statement is the hero here, (outside the function) we have written `puts x` after `print_x`, if you expected to print 3 then you are wrong. Here `x` is the `x` that we have declared it outside the function, here it will get printed as 5. This means that a variable declared inside the function has no scope outside it.

To know more and to convince ourself that variable declared inside a function has no scope outside it, we will try out another program. Type in the program variable_scope_2.rb into your text editor and execute it.

```
#!/usr/bin/ruby
# variable_scope_2.rb

def print_variable
      y = 3
      puts y
end

print_variable
puts y
```

Output

```
3
variable_scope_2.rb:10: undefined local variable or method `y' for main:Object
(NameError)
```

Here is how the program works or here is how the program doesn't works as it throws an error. Take a look at the function `print_variable`, in it we have declared a variable called `y` using statement `y = 3` and told the ruby interpreter to print its value using statement puts `y`. So in the program when we call `print_variable` the `y` is declared and its value 3 is printed without a glitch. Next we say `puts y` outside the function `print_variable`, since `y` is only declared within the function outside it, the variable `y` doesn't exist and in technical terms it has no scope, so the Ruby interpreter throws an error. So we get the error message as follows:

```
variable_scope_2.rb:10: undefined local variable or method `y' for main:Object
(NameError)
                                    ---- x X x ----
```

Looks like Matz (the creator of Ruby) hasn't seen the movie Back to the future. Lets see another program that proves that time travel isn't built into Ruby, type the program below (*variable_scope_3.rb*) into your text editor and execute it.

```
#!/usr/bin/ruby
# variable_scope_3.rb

puts a # you can't access a variable that will be created in future
a = 10
```

Output

```
variable_scope_3.rb:4: undefined local variable or method `a' for main:Object
(NameError)
```

If you have anticipated right, the program throws out an error. We have given *puts a* before *a* has been declared. Ruby interpreter does not consider whats declared in future, so when *puts a* is encountered it, at that point of time *a* is undeclared, and hence an error is thrown. In other words scope of an variable starts only after it has been declared.

Global Variables

If you are a one who don't like the idea that variables declared outside a function cant be accessed from it, then Ruby provides a way to do it. There are special variables called global variables that can be accessed from any where. Global variables are preceded by a dollar ($) sign. To know about global variables lets see an example. Type the program below (global_variables.rb) and execute it.

```
#!/usr/bin/ruby
# global_variables.rb

$x = 5

def print_x
      $x = 3
      puts $x
end

print_x
puts $x
```

Output

```
3
3
```

Having run it successfully lets see how it works. First we declare a global variable *$x* and assign it to the value *5* in the statement *$x = 5*. Next we define a function *print_x* in which we change the value of *$x* to *3* using statement *$x = 3*, then we print the value of *$x* using *puts $x*. So obviously

we call *print_x* we get the output as *3*. Next outside the function after calling *print_x*, we print the value of *$x* using *puts $x*. If you think it would print *5*, then you are mistaken. Since *$x* can be accessed from any where, and we have called *print_x*, in *print_x* we have changed the value of *$x* to *3*, no matter what, even outside the scope of the function the value of *$x* will be changed.

Lets see another example to understand global variables better. Look at the example below (*global_variables_1.rb*), type it in your text editor and execute it

```
#!/usr/bin/ruby
# global_variables_1.rb

$x = 5

def print_x
      puts $x
end

print_x
$x = 7
print_x
$x = 3
print_x
```

here is how the output of the program looks like

```
5
7
3
```

Lets see how the program works. At first we declare a global variable *$x* and assign it to value five using *$x = 5*, then we define a function called *print_x* in which we just print out the value of *$x* using *puts $x* statement. While we call the first *print_x* statement, the value of *$x* is 5 and hence 5 gets printed. Next we change the value of *$x* to 7 in statement *$x = 7* and when we call *print_x*, the value of *$x* which is now 7 gets printed. Finally we set *$x* to 3 using *$x = 3*, when we call *print_x* for the final time 3 gets printed out.

This program proves that global variables can be manipulated from any where and these manipulated values can be accessed from any where.

Next arises a question weather global variable and local variable can have the same name. The answer is yes. Its because global variables start with a $ sign and local variables start with a letter or underscore character, so ruby can clearly tell the difference between them. Lets see a program that proves this, read, learn type and execute the program given below (global_variables_2.rb). Once you are done with it, we will see how it works.

```
#!/usr/bin/ruby
# global_variables_2.rb
```

```
$x = 5
x = 5

def print_x
      $x = 3
      x = 3
      puts "In print_x"
      puts "$x = "+$x.to_s
      puts "x = "+x.to_s
end


print_x
puts "Outside print_x"
puts "$x = "+$x.to_s
puts "x = "+x.to_s
```

Output

```
In print_x
$x = 3
x = 3
Outside print_x
$x = 3
x = 5
```

In the above program we declare two variables one global `$x` and assign it to value 5 and another local `x` and assign it to value 3 in the following lines

```
$x = 5
x = 5
```

Next we create a function `print_x` in which we change the value of `$x` to 3, since `$x` is global, the change is affected every where in the program, next we have statement `x =3`, this variable `x` is local one and is different from `x = 5` which we defined outside the function. Next we will tell the program to print the values of `$x` and local `x` using he following statements

```
puts "$x = "+$x.to_s
puts "x = "+x.to_s
```

OK, when the program encounters the `print_x` call, we get the following output

```
In print_x
$x = 3
x = 3
```

Note that `$x` is now 3 and local `x` is also 3. Now outside the function we print the values of `$x` and `x` using the following statements (last 3 lines of the program)

```
puts "Outside print_x"
puts "$x = "+$x.to_s
puts "x = "+x.to_s
```

When these statements are executed, we get the following output

```
Outside print_x
$x = 3
x = 5
```

Here as `$x` has been assigned to 3, 3 is printed as its value. `x` over here remains 5 as here `x` refers to not the `x` thats defined inside `print_x`, but the one thats defined out of it.

Classes & Objects

Creating a Square

Classes can be thought as variables and functions bundled under one name. To illustrate about classes lets look at a humble object called square. Square is a very simple geometric shape that has four sides of equal length, how to represent this square in a Ruby program? We now write an empty class called square

```
#square.rb

class Square
end
```

We have written an empty class. The word *class* tells that we are writing a definition of a class. What follows the class keyword is the name of the class, in this case is *Square*. One must note that name of a class in Ruby must start with a capital letter.

A square has got four sides, all have the same length. We now put an variable into square called *side_length*.

```
#square.rb

class Square
      attr_accessor :side_length
end
```

You might ask what *attr_accessor* is, it stands for attribute accessors, which enables you to get and set the *side_length* easily. To continue a shallow learning curve lets stop here. Lets use this square class and see how it works. Modify the code above as shown below:

```
#square.rb

class Square
      attr_accessor :side_length
end

s1 = Square.new # creates a new square
s1.side_length = 5 # sets its side length
puts "Side length of s1 = #{s1.side_length}" # prints the side length
```

When you execute the above code, this is what you will get as result

```
Side length of s1 = 5
```

Lets walk thru the newly added code, take the line

```
s1 = Square.new
```

In the above statement we create a new square and store its parameters into a variable called *s1*. A new class instance can be created by using *<class name>.new*. Having created a new square, we can now access its *side_length* using the dot operator '.'. So we first set the *side_length* to five units using the following statement

```
s1.side_length = 5
```

Now having assigned the side length, we can use it for any purpose. Now we just simply print the *side_length* of square using the following statement;

```
puts "Side length of s1 = #{s1.side_length}"
```

Functions in Class

We have a class called square which has a attribute named *side_length*. With the *side_length* we can find the squares area, its perimeter and its diagonal length. In this example I am going to find the area and perimeter. Why I omit diagonal length? Well its my wish, if you are not happy about it modify this book and redistribute it. So lets add two functions to find our area and perimeter. Modify the code as shown (I am saving the modified code in a file called square_1.rb)

```
#square_1.rb

class Square
      attr_accessor :side_length

      def area
            @side_length * @side_length
      end

      def perimeter
            4 * @side_length
      end
end
```

In the highlighted code above you see that I have added two functions, one named *area* and another named *perimeter* which computes and returns the area and perimeter of the square respectively . These functions are very similar to any other function we have created before, only now its placed inside a class. Lets write some additional code to exploit the new features we have added, just add the code highlighted below and run it

```
#square_1.rb

class Square
      attr_accessor :side_length

      def area
            @side_length * @side_length
      end
```

```
      def perimeter
            4 * @side_length
      end
end

a = Square.new
a.side_length = 5
puts "Area: #{a.area}"
puts "Perimeter: #{a.perimeter}"
```

Run the example and here is what you will get as output

```
Area: 25
Perimeter: 20
```

The explanation is pretty straight forward in the following lines

```
a = Square.new
a.side_length = 5
```

We have declared a new square and have assigned *side_length* as *5* units. In lines below we simply print out the values of *a.area* and *a.perimeter*

```
puts "Area: #{a.area}"
puts "Perimeter: #{a.perimeter}"
```

See how we have embedded the values of a's area and perimeter (highlighted in the code above). One thing must be irking you if you are reading this book right, see the function area:

```
def area
      @side_length * @side_length
end
```

We know that square has an attribute called side_length which is defined by the statement *attr_accessor :side_length* well as shown in highlighted code above we have used *@side_length* instead of *side_length* , thats because inside the class, class-variables are prefixed with @ (at) symbol. This helps us to identify between class variables and local variables or functions that share the same name.

Initializers or Constructors

In previous examples where we dealt with squares, have you ever wondered what happens when you say like *s = Square.new* ? Well in this case a new *Square* gets created and its put inside the variable *s*. If one asks a question weather we can do something when a Square initializes? The answer is yes! In fact you can do almost anything you want. All you have to do is to put code inside a function called initialize, this function gets called when ever there is a <class name>.new call

Look at the example *square2.rb* , take a good look at the highlighted or darkened lines, there we define a function called *initialize* , this function takes in one argument named *side_length*

who's default value is zero. If *side_length* is specified, it sets the *@side_length* attribute in the square class to that value else *@side_length* takes the default value of zero. Type square2.rb into text editor and execute it

```
#square_2.rb

class Square
      attr_accessor :side_length

      def initialize side_length = 0
            @side_length = side_length
      end

      def area
            @side_length * @side_length
      end

      def perimeter
            4 * @side_length
      end
end

s1 = Square.new 4
s2 = Square.new
s2.side_length = 5

puts "Area of s1 is #{s1.area} squnits"
puts "Peimeter of s2 is #{s2.perimeter} units"
```

Output

```
Area of s1 is 16 squnits
Perimeter of s2 is 20 units
```

In the program concentrate on the following lines

```
s1 = Square.new 4
s2 = Square.new
s2.side_length = 5
```

In the first line *s1 = Square.new 4* we create a Square named *s1* who's *@side_length* is 4 units. In the second line *s2 = Square.new* we create a *Square* named *s2*, initially its side length (*@side_length*) would be set to zero units, only in the third line *s2.side_length = 5* its *@side_length* is set to *5* units.

In rest of *the* code

```
puts "Area of s1 is #{s1.area} squnits"
puts "Peimeter of s2 is #{s2.perimeter} units"
```

We print the area of Square *s1* and perimeter of Square *s2* which produces the *desired* output.

Private Methods

By default the methods or functions in a class is public (can be accessed outside the classes scope),

if you don't want it to be accessed by programs *outside* a class you can make it private. Lets create a class called Human, and have a private method, lets try to access it from outside the class and see what happens to it. Type in the program and execute it

```
# private_method.rb

class Human
      attr_accessor :name, :age

      def tell_about_you
            puts "Hello I am #{@name}. I am #{@age} years old"
      end

      private

      def tell_a_secret
            puts "I am not a human, I am a computer program. He! Hee!!"
      end

end

h = Human.new
h.name = "Zigor"
h.age = 314567
h.tell_about_you
h.tell_a_secret # this wont work
```

The program above when executed produces the following result

```
Hello I am Zigor. I am 314567 years old
human.rb:20: private method `tell_a_secret' called for #<Human:0xb7538678
@name="Zigor", @age=314567> (NoMethodError)
```

Look at the highlighted line in the program, the function *tell_a_secret* is placed under the keyword *private*, this makes it not accessible from outside the class. Note the line when we call the method *tell_a_secret* , it throws an error, in fact it says a no method error (*NoMethodError*) which means that the called method does not exist in the class. It does not mean the computer is telling a lie, instead its safely keeping a secret.

In programing you only let certain parts of your program visible to others, this helps keep the interface simple and give your users only the resource they really need to write code, this sort of hiding unwanted things uncomplicates programming.

One might question, if there is no way to access a private method, then why we need to have it? Well there are ways to access it indirectly as you see in example below. Type it and execute it

```
# private_method_1.rb

class Human
      attr_accessor :name, :age
```

```
      def tell_about_you
            puts "Hello I am #{@name}. I am #{@age} years old"
      end

      def confess
            tell_a_secret
      end

      private

      def tell_a_secret
            puts "I am not a human, I am a computer program. He! Hee!!"
      end


end


h = Human.new
h.name = "Zigor"
h.age = 314567
h.tell_about_you
h.confess
```

This is how the result will be

```
Hello I am Zigor. I am 314567 years old
I am not a human, I am a computer program. He! Hee!!
```

Take a good look at the method *confess* , in it we call the private method *tell_a_secret* , so when we call *confess* even outside the class (*h.confess*), the confess method which is public calls the private method, since *confess* is inside the class *Human*, it can access any private method without an hindrance, so the program executes perfectly.

Class variables and methods

Till now we have learned to create a class, we know that a class can have certain attributes, for example a human might have attributes like name, age and blah blah... We know that class can have some functions in it which can be called by variable which is a instance of the class. OK fine well and good. What if we want to call a function or a class without declaring a variable which is the instance of that class? Those functions that can be called without declaring a instance variable that belongs to the class type is called class methods. Those variables that can be accessed without using instance variables are called class variables.

Lets look at a program that will demonstrate class variables and methods. In the following program *class_var_methods.rb* , I will create a class named *Robot*. It will have a class variable named *@@robot_count* which will keep track of how many Robots that were created. Since its a class

variable we indicate it to the computer by using two @ (at) symbols before it, hence in program we denote it like *@@robot_count* .

We create a function named *robots_created* that will return number of Robots that were created. Notice (in program below) that the function *robots_created* is *written* as *self.robots_created* , the self keyword tells the computer that this function can be called without an instance object being declared.

Type the program shown below *class_var_method.rb* in text editor and execute it

```
# class_var_methods.rb

class Robot
      def initialize
            if defined?(@@robot_count)
                  @@robot_count += 1
            else
                  @@robot_count = 1
            end
      end

      def self.robots_created
            @@robot_count
      end
end

r1 = Robot.new
r2 = Robot.new
puts "Created #{Robot.robots_created} robots"
r3, r4, r5 = Robot.new, Robot.new, Robot.new
puts "Created #{Robot.robots_created} robots"
```

When executed the program above will give the following *result*

```
Created 2 robots
Created 5 robots
```

Lets see the *initialize* method and lets analyze how we keep *track* of number of Robot's created. When the first Robot is created in the statement

```
r1 = Robot.new
```

The program control goes to the *initialize* method, since its the first *time* the variable *@@robot_count* is not defined, so the condition in the following *if* statement

```
            if defined?(@@robot_count)
                  @@robot_count += 1
            else
                  @@robot_count = 1
            end
```

fails and the code goes to the *else* part and there *@@robot_count = 1* defines *the* variable *@@robot_coun*t and initializes to value *1*.

In the second statement where we create robot named *r2* using the following command

```
r2 = Robot.new
```

The control once again goes to the *initialize* method, there the *if* statement passes as *@@robot_count* has already been defined when we created *r1*, now the *@@robot_count* gets incremented by 1, now becomes 2.

Next is the puts statement we call *Robot.robots_created* which just returns the *@@robot_count* , hence 2 gets printed. Next in the following statement:

```
r3, r4, r5 = Robot.new, Robot.new, Robot.new
```

we create three new robots *r3*, *r4* and *r5*. Now the *@@robot_count* will get incremented to *5*. In the next *puts* statement, the result gets printed. The moral of the story is this, class methods have a *self.* (self dot) before them, class variables have two @ (@@) before them.

Fine, hope everything went right for the reader. Why now we use *attr_accessor* for *robot_count* variable so that our program gets simplified as shown below. Type in and execute it.

```
# attr_for_classvar.rb
# this program dosent work

class Robot
      attr_reader :robot_count

      def initialize
            if defined?(@@robot_count)
                  @@robot_count += 1
            else
                  @@robot_count = 1
            end
      end
end

r1 = Robot.new
r2 = Robot.new
puts "Created #{Robot.robot_count} robots"
r3, r4, r5 = Robot.new, Robot.new, Robot.new
puts "Created #{Robot.robot_count} robots"
```

What was the result you got? Can *attr_accessor* be used for class variables?

Inheritance

We evolved from monkeys. Chimps look like us, we both share many characteristics, we have many attributes similar to chimps, they are so similar us, that chimps were sent into space before us to see the impact of zero gravity on a monkeys body. Only when the scientist felt safe did they send humans[19]. When man evolved from monkeys he inherited many things from them, for example we

19 Uri Gagarin of the Soviet union was the first man to go into space

look like monkeys, don't believe it? Just go and stand in front of a mirror!

OK, in programing world we have a thing called inheritance in which one class can have property of another class with some little (or sometimes extreme) changes. Let me tell you a math truth, 'a square is a rectangle in which all sides are equal' , is it not so? All squares are rectangles, but not all rectangles are squares. We will be using this stuff to write our next program *inheritance.rb* . Write the program in text editor and execute it.

```
# inheritance.rb

class Rectangle
      attr_accessor :length, :width

      def initialize length, width
            @length = length
            @width = width
      end

      def area
            @length * @width
      end

      def perimeter
            2 * (@length + @width)
      end
end

class Square < Rectangle

      def initialize length
            @width = @length = length
      end

      def side_length
            @width
      end

      def side_length=(length)
            @width = @height = length
      end
end

s = Square.new 5
puts "Perimeter of the square s is #{s.perimeter}"
r = Rectangle.new 3, 5
puts "Area of rectangle r is #{r.area}"
```

When executed, the program above produces the following result

```
Perimeter of the square s is 20
Area of rectangle r is 15
```

Read the program carefully, we have defined a class called *Rectangle* that has two attributes namely *@length* and *@width*. When we initialize it in statement *r = Rectangle.new 3, 5*, we pass these two parameters. When *area* is called the product of these two attributes is returned, when

its perimeter is called using some genius formula the perimeter is calculated and returned. Fine , we then define a class called *Square* that inherits the properties of *Rectangle*. To say that the class *Square* inherits *Rectangle* we use a < (less than) sign as shown

```
class Square < Rectangle
```

Take a look at the initialize method in *Square* class, it takes only one argument length which it uses to set the values of attributes *@length* and *@width* . Since *Square* inherits *Rectangle* class it gets all attributes and methods of *Rectangle* by default. Having set the *@width* and *@height* of the *Square* we can now call the Square's *area* and *perimeter* functions just like we do that with *Rectangle*.

Overriding Methods

We have seen that class can inherit attributes and methods from its base class. Lets say that we have a class A that is a parent class of B (i.e. B inherits A), now the scenario is there is a method defined in B which has the same name that of method in A. When we create a instance variable of type B and call that method name it will check if the method is present in class B if yes that method will be executed, if that method is not found in B, then the Ruby interpreter checks it in class A, if its found its executed, else NoMethodError[20] is raised.

To make this clear lets see an example, type and execute *override_methods.rb*

```
#!/usr/bin/ruby
# override_methods.rb

class A
      def belongs_to
            puts "I belong to in class A"
      end

      def another_method
            puts "Just another method in class A"
      end
end

class B < A
      def another_method
            puts "Just another method in class B"
      end
end


a = A.new
b = B.new
a.belongs_to
```

20 No method error means that the method name called cannot be found, hence an error is raised by the Ruby interpreter

```
a.another_method
b.belongs_to # This is not overriden so method in class A is called
b.another_method # This is overridden so method in class B is called
```

Result

```
I belong to in class A
Just another method in class A
I belong to in class A
Just another method in class B
```

Take a look at the result. When *a.belongs_to* is called the program prints out *I belong to in class A* as it was defined in class A. When *a.another_method* is called we see the program prints out *Just another method in class A* as it was defined in class A . When *b. belongs_to* is called the program once again prints out *I belong to in class A* as there is no *belongs_to* method in class B and hence the parent method is called. See the drama when *b.another_method* is called , the program prints out *Just another method in class B* and not *Just another method in class A* as B has *another_method* in its scope, so there is no need to look for that method in class A.

We will take the concept of overriding a step further, know that everything in Ruby is a object. Ruby is purely an object oriented programing language. Shoot up your irb and type the following

```
>> "Something".class
=> String
>> 1.class
=> Fixnum
>> 3.14278.class
=> Float
```

When ever in Ruby you put an *.class* after object it returns the class to which the object belongs. So we see that numbers like 1, 2 ,3..... belong to the *Fixnum* class. Lets override its key method the + . Plus sign is used in Ruby to add two numbers. Try these examples in your irb

```
>> 1+2
=> 3
>> 478+90
=> 568
```

So we see that when there is a Fixnum, followed by a plus sign and another Fixnum Ruby interpreter adds these two Fixnum's which is returned as result. Lets now mess up with the plus sign. Take a look at override_methods_1.rb , type it in your text editor and execute it.

```
# override_methods_1.rb

class Fixnum
      def + a
            416
      end
end
```

```
puts 3+5
puts 7+12
```

Result

```
416
416
```

Look at the result, aren't you surprised? When three and five are added the result you get is 416 and when 7 and 12 are added, once again the result is 416. Take a look at the blackened or highlighted code in the Fixnum class. To make your reading convenient, here is the code:

```
def + a
      416
end
```

In it we have redefined the method + in Fixnum class. In it, we have said no matter what ever be the value of a (that is number that is at the right side of + sign in addition,) we must return a value of 416, so the Ruby interpreter simply obeys it.

Fixnum is a core class of Ruby, in many programming languages (for example Java) , one does not have the luxury to modify the core class as Ruby allows it to do. Many authors and programming gurus who have written books about Ruby have called this Ruby feature as a dangerous one, its dangerous indeed, if you do modify a important class in Ruby and if our code is buried deep in a project, some times it can result in severe logical errors in our program and sometimes may cost lot of resource waste as one needs to bury himself to debug the code. So before overriding methods in a important or core class please think, and then do make a leap.

Extending class

Ruby lets the programmer to extend preexisting classes in (almost) any way you want, it doesn't matter if the classes are written by you or bundled into the Ruby language itself. In the following example we will be extending Fixnum class to suit our needs. Type the program into text editor and execute it

```
# extending_class.rb

class Fixnum
      def minute
            to_s.to_i * 60
      end

      def hour
            to_s.to_i.minute * 60
      end

      def day
            to_s.to_i.hour * 24
```

```
      end

      def week
            to_s.to_i.day * 7
      end
end


puts Time.now + 2.week
```

Result

```
Wed Apr 07 16:55:44 +0530 2010
```

The program puts what would be Time exactly 2 weeks from current second, note that we do it by this statement :

```
puts Time.now + 2.week
```

The *Time.now* gets the current Time instance, to which we add *2.week*. In reality, the native Fixnum class has no method named *week* in it but see in the program we have defined a Fixnum class which has method name *week*, when its called it returns number of seconds in a week which can be added or subtracted to time object to get past and future time.

In a similar fashion you can extend any class in Ruby, you can override almost any method. Of course some programmers (who like to be called as professionals) see this as a threat as some accidental changes might introduce a bug in your code, but if you truly love Ruby this shouldn't matter a lot.

Reflection

Reflection is a process by which a computer program can analyze itself and modify it on the go. In the following pages we will just be scratching its surface. If time willing I will add meta programming section in this book where I can discuss more. Right now I don't really know what meta programming is.

OK, so lets jump in. We will try out these examples in irb , so in your terminal type irb –simple-prompt . In the irb prompt below I have declared a String variable *a* and set it to a value *"Some string"*

```
>> a = "Some string"
=> "Some string"
```

Now lets see that what methods are available with variable that we can use. To do so type *a.methods* in irb

```
>> a.methods
=> ["upcase!", "zip", "find_index", "between?", "to_f", "minmax", "lines",
```

```
"sub", "methods", "send", "replace", "empty?", "group_by", "squeeze", "crypt",
"gsub!", "taint", "to_enum", "instance_variable_defined?", "match", "downcase!",
"take", "find_all", "min_by", "bytes", "entries", "gsub", "singleton_methods",
"instance_eval", "to_str", "first", "chop!", "enum_for", "intern", "nil?",
"succ", "capitalize!", "take_while", "select", "max_by", "chars", "tr!",
"protected_methods", "instance_exec", "sort", "chop", "tainted?", "dump",
"include?", "untaint", "each_slice", "instance_of?", "chomp!", "swapcase!",
"drop", "equal?", "reject", "hex", "minmax_by", "sum", "hash",
"private_methods", "all?", "tr_s!", "sort_by", "chomp", "upcase", "start_with?",
"unpack", "succ!", "enum_slice", "kind_of?", "strip!", "freeze", "drop_while",
"eql?", "next", "collect", "oct", "id", "slice", "casecmp", "grep", "strip",
"any?", "delete!", "public_methods", "end_with?", "downcase", "%", "is_a?",
"scan", "lstrip!", "each_cons", "cycle", "map", "member?", "tap", "type", "*",
"split", "insert", "each_with_index", "+", "count", "lstrip", "one?",
"squeeze!", "instance_variables", "__id__", "frozen?", "capitalize", "next!",
"each_line", "rstrip!", "to_a", "enum_cons", "ljust", "respond_to?", "upto",
"display", "each", "inject", "tr", "method", "slice!", "class", "reverse",
"length", "enum_with_index", "rpartition", "rstrip", "<=>", "none?",
"instance_variable_get", "find", "==", "swapcase", "__send__", "===", "min",
"each_byte", "extend", "to_s", "rjust", "index", ">=", "size", "reduce", "tr_s",
"<=", "clone", "reverse_each", "to_sym", "bytesize", "=~",
"instance_variable_set", "<", "detect", "max", "each_char", ">", "to_i",
"center", "inspect", "[]", "reverse!", "rindex", "partition", "delete", "[]=",
"concat", "sub!", "dup", "object_id", "<<"]
```

As you can see *a.methods* returns the methods / functions that can be used upon a String. Next we will try out and find what class or type *a* belongs. Of course we know that it belongs to String type, but for the sake of learning to program type a.class into irb

```
>> a.class
=> String
```

and it faithfully returns that a is of the type String.

Fine we have seen some thing about reflection till now. To understand it better lets define our own class and see how reflection works upon it. Type the program (below) reflection.rb into a text editor and execute it.

```
# reflection.rb

class Someclass
      attr_accessor :a, :b

      private
      # A dummy private method
      def private_method
      end

      protected
      # A dummy protected method
      def protected_method
      end

      public
      # A dummy public method
      def public_method
      end
end
```

```
something = Someclass.new
something.a = 'a'
something.b = 123
puts "something belongs to #{something.class}"
puts
puts "something has the following instance variables:"
puts something.instance_variables.join(', ')
puts
puts "something has the following methods:"
puts something.methods.join(', ')
puts
puts "something has the following public methods:"
puts something.public_methods.join(', ')
puts
puts "something has the following private methods:"
puts something.private_methods.join(', ')
puts
puts "something has the following protected methods:"
puts something.protected_methods.join(', ')
```

Output

```
something belongs to Someclass

something has the following instance variables:
@a, @b

something has the following methods:
inspect, protected_method, tap, clone, public_methods, __send__, object_id,
instance_variable_defined?, equal?, freeze, extend, send, methods,
public_method, hash, dup, to_enum, instance_variables, eql?, a, instance_eval,
id, singleton_methods, a=, taint, enum_for, frozen?, instance_variable_get,
instance_of?, display, to_a, method, b, type, instance_exec, protected_methods,
==, b=, ===, instance_variable_set, kind_of?, respond_to?, to_s, class, __id__,
tainted?, =~, private_methods, untaint, nil?, is_a?

something has the following public methods:
inspect, tap, clone, public_methods, __send__, object_id,
instance_variable_defined?, equal?, freeze, extend, send, methods,
public_method, hash, dup, to_enum, instance_variables, eql?, a, instance_eval,
id, singleton_methods, a=, taint, enum_for, frozen?, instance_variable_get,
instance_of?, display, to_a, method, b, type, instance_exec, protected_methods,
==, b=, ===, instance_variable_set, kind_of?, respond_to?, to_s, class, __id__,
tainted?, =~, private_methods, untaint, nil?, is_a?

something has the following private methods:
exit!, chomp!, initialize, fail, print, binding, split, Array, format, chop,
iterator?, catch, readlines, trap, remove_instance_variable, getc,
singleton_method_added, caller, putc, autoload?, proc, chomp, block_given?,
throw, p, sub!, loop, syscall, trace_var, exec, Integer, callcc, puts,
initialize_copy, load, singleton_method_removed, exit, srand, lambda,
global_variables, gsub!, untrace_var, open, `, system, Float, method_missing,
singleton_method_undefined, sub, abort, gets, require, rand, test, warn, eval,
local_variables, chop!, scan, raise, printf, set_trace_func, private_method,
fork, String, select, sleep, gsub, sprintf, autoload, readline, at_exit,
__method__

something has the following protected methods:
protected_method
```

You must have got pretty big output as shown above. Lets now walkthru the code. First we define a

class called *Someclass* in which we have two attributes / class variables *a* and *b*. We have a private method called *private_method* , protected method called *protected_method* and public method called *public_method*.

After defining the class we create a variable called *something* of the type *Someclass* and give values to its attributes in the following lines

```
something = Someclass.new
something.a = 'a'
something.b = 123
```

Next we ask the ruby interpreter to print the class of variable something using the following statement *puts "something belongs to #{something.class}"* which it faithfully does and so we get the following output:

```
something belongs to Someclass
```

Next we would like to know that if *something* which is an object of type *Someclass* has any instance variables. To know it we command as

```
puts "something has the following instance variables:"
puts something.instance_variables.join(', ')
```

for which we get the following output

```
something has the following instance variables:
@a, @b
```

Next we would like to know what methods are there with something that can be used. To know that we can use the methods function, so we write the following code:

```
puts "something has the following methods:"
puts something.methods.join(', ')
```

In the above code *something.methods* returns an array of methods, this must be converted to a string which is done by the *join* method. The elements of the array are joined by the String passed to the *join* method. Notice that there are more methods than we have defined, thats because even *Someclass* is of type *Object*[21] which itself has many methods of its own. In Ruby everything is a Object!

The methods and *public_methods* of any Object returns the same result. So we will skip the discussion on these

```
puts "something has the following public methods:"
puts something.public_methods.join(', ')
```

statements

21 Object is one of the most fundamental class in Ruby upon which almost all other classes are built upon. All classes implicitly inherit Object

Next we want to know what are the private, public and protected methods are, that are in *Someclass*, since *something* belongs to *Someclass*, private methods can be got using *private_methods* function, thus by giving the following statements

```
puts "something has the following private methods:"
puts something.private_methods.join(', ')
```

We are able to get private methods in some class.

Similarly protected methods are got by using *protected_methods* function which I won't discuss due to my laziness.

Encapsulation

You must have taken a capsule tablet in some point of time in your life. In it the medicine is packed inside a gelatin capsule. When you take it with water it slides to your stomach where water breaks out the gelatin layer releasing the medicine in it which cures your body of ailments. If you try to swallow the medicine without the capsule it will be a bitter experience.

In similar fashion modern programing language allows you to hide unwanted details and let your fellow programmer look only at the needed details. This technique is called encapsulation which when properly implemented will result in producing clean code and one thats easy to use.

Another great example of encapsulation is your car. Under the hood your car has thousands of parts that turn this way and that way, yet all you need to do is to turn on the key and operate the steering wheel and pedals[22] to get a driving experience. There is no need for you to bother what goes on behind the hood.

Lets see a small example that will explain to us how encapsulation works. Type out the program *encapsulation.rb* in your text editor and execute it.

```
# encapsulation.rb

class Human
      attr_reader :firstname, :lastname

      def name=(name)
            @firstname, @lastname = name.split
      end
end

guy = Human.new
guy.name = "Ramanuja Iyengaar"
puts "First name: #{guy.firstname}"
puts "Last name: #{guy.lastname}"
```

22 Gears are slowly being eliminated in todays cars

Output

```
First name: Ramanuja
Last name: Iyengaar
```

So we get the first name of the person as Ramanuja and last name as Iyengar. These two lines are printed out due to the following statements

```
puts "First name: #{guy.firstname}"
puts "Last name: #{guy.lastname}"
```

See the two lines before these statements. First we declare a new variable named guy of the type human by writing `guy = Human.new` , next we set `guy.name = "Ramanuja Iyengaar"` , but in the first puts statement we call `guy.firstname` and in the next one we call `guy.lastname` and we get the answers! This is because inside the program in the method called `name` (see highlighted code) we split it and assign the word before space as `@firstname` and word after space as `@lastname` using the following piece of code:

```
@firstname, @lastname = name.split
```

So when we call `guy.firstname` and `guy.lastname` it gets printed faithfully. Note that outside the class we never set the `@first_name` and `@last_name`, it was totally encapsulated from us.

One might be wondering what the statement attr_reader :firstname, :lastname does? It declares two variables `@first_name` and `@last_name`. The attr_reader signifies that the two variables can only be read by program outside the class, in other words if we try to set `guy.first_name = "Ramanuja"` the Ruby interpreter will throw out an error.

Polymorphism

Poly means many, and morphis means forms. I think its either in Greek or Latin, who cares? In programming language you can use one thing to do many things, lets see a few examples. Lets take the humble plus sign. When we take “Hello ” and “World!” and put a plus sign in between, the output is “Hello World!”. In technical talk we call this concatenation (joining together). Why there are so many difficult words to learn when you want to be programmer? Who knows? Possibly speaking some non-understandable blah blah might make you look intelligent, possibly fetch higher pay. So here is the irb example:

```
>> "Hello " + "World!"
=> "Hello World!"
```

Now lets use this plus sign on numbers. We now stick it between 134 and 97. When we do that we get the answer as 231 and not as 13497. Why? Its because the plus sign is trained to do different

things when its stuck in between different things.

```
>> 134 + 97
=> 231
```

When you stick it in between String's[23] it joins them, when you stick it in between numbers it adds them. So the operator plus takes many forms or does different operations depending upon the situation.

In a similar way what will happen if we multiply a string by a number. Well when we do it as shown below

```
>> "Hello"*5
=> "HelloHelloHelloHelloHello"
```

we see that string is printed the number of times. So multiplying “Hello” by 6 prints “Hello” six times. In the next example we assign value six to a variable named hello and multiply it by five

```
>> Hello = 6
=> 6
>> Hello*5
=> 30
```

Since hello is a variable that carries a number, multiply it, results is a number. So you see even an multiplication operator takes many forms or different functions depending on the situation. Its like this a policeman when at home is kind with his family, when he is made to take a thud he behaves in a different way.

In a similar way the length operator / function, when you are finding out the length of a string, it tells the number of characters in it.

```
>> "some text".length
=> 9
```

When you are finding the length of an array it tells the number of elements the array has.

```
>> [5, 7, "some text", Time.now].length
=> 4
```

So we see that in Ruby, a thing can do different things, just like an real world object does[24].

Class Constants

Just like any other programming language, one can have a constant values in a class in Ruby. Lets jump into action, take a look at the following program *class_constant.rb* , it source code is like this

23 A piece of text

24 Like nuclear energy can be used to explode an atomic bomb or power an entire city, depending on what humans decide to do with it

```
#!/usr/bin/ruby
# class_constant.rb

class Something
      Const = 25
end

puts Something::Const
```

Output

```
25
```

Note the pieces of code I have darkened. In the class Something, you see the statement *Const = 25*. If you were reading this book well, you might realize that constant in Ruby starts with a capital letter. In the class Something, we have declared a constant names *Const* and assigned it to *25*.

Note the statement *puts Something::Const* . *puts* is for printing almost anything thrown at it. Here we throw *Something::Const* and it faithfully prints out the constant value. So class constants can be access by *<class_name>::<constant_name>* , this is how you access constants of a class.

Let's see how class instance can access a class constant. Type the program *class_constant_1.rb*

```
#!/usr/bin/ruby
# class_constant.rb

class Something
      Const = 25
end


puts Something::Const
s = Something.new
puts s.Const
```

Output

```
25
class_constant_1.rb:10: undefined method `Const' for #<Something:0xb745eb58>
(NoMethodError)
```

So in this program (above) we have declared a variable *s* whose class is *Something*. In line *puts s.Const* , we try to access the constant value inside something via its instance variable *s* and we get a No Method Error, or the Ruby interpreter thinks *Const* is a method since we use *s* dot *Const*. To fix this issue, you can write a method called *Const* and call it as shown in *class_constant_2.rb*

```
#!/usr/bin/ruby
# class_constant_2.rb

class Something
      Const = 25

      def Const
            Const
      end
end
```

```
puts Something::Const
s = Something.new
puts s.Const
```

Output

```
25
25
```

So defining a method[25] and returning *Const* from it solves the problem.

Some might think one can access class constant value using the instance variable by using double colon (::) instead of the dot operator as shown in *class_constant_3.rb* , well it wont work as you can see from its output

```
#!/usr/bin/ruby
# class_constant_3.rb

class Something
      Const = 25

      def Const
            Const
      end
end

puts Something::Const
s = Something.new
puts s::Const
```

Output

```
25
class_constant_3.rb:14: #<Something:0xb74029fc> is not a class/module (TypeError)
```

Modules and Mixins

When ever you think of modules, you think of a box or something. Just look at your printer. Its a module. It can do some thing. It has things to do something. In a similar way modules in Ruby can contain Ruby code to do something. Modules are way to pack Ruby code into possible logic units. When ever you want to use the code in a module, you just include it in your Ruby program.

25 It's not necessary that the method should have the same name as the constant.

Lets look at our first module program called *module_function.rb* . The program below has two modules namely *Star*, *Dollar*. Both these modules have the same function called *line*. Note that in the function *line* in module *Star*, we print a line of 20 star (*) characters. In similar fashion in function *line* in module *Dollar* we print a line of 20 dollar ($) characters. Type the program in your text editor and execute it.

```
# module_function.rb

module Star
      def line
            puts '*' * 20
      end
end

module Dollar
      def line
            puts '$' * 20
      end
end

include Star
line
include Dollar
line
```

Output

```
********************
$$$$$$$$$$$$$$$$$$$$
```

Lets look at the program thats outside the module. We have the following code:

```
include Star
line
include Dollar
line
```

In the line *include Star*, we include the code thats in the *Star* module, then we call the function *line*, so we get output as a line of 20 stars. Look at the next line, we include the module *Dollar*. *Dollar* too has a function called *line*. Since this module is called after *Star*, the *line* function in *Dollar* module over writes or in the right terms hides the *line* function in the *Star* module. Hence calling line after *include Dollar* will execute code in the *line* function of *Dollar* module. Hence we get a line of twenty dollar sign.

In the coming example *module_function_0.rb* we will see what happens when we call the line

function without including any module. Type the program below and execute it

```
# module_function_0.rb

module Star
      def line
            puts '*' * 20
      end
end

module Dollar
      def line
            puts '$' * 20
      end
end

line
```

Output

```
module_function_0.rb:15:in `<main>': undefined local variable or method `line'
for main:Object (NameError)
```

As you can see that *line* is considered as undefined local variable or a method[26]. So we can say that the functions in module can be accessed only if the module is included in your program.

Lets say that we write another module without any function but just code in it. I wrote the following program *module.rb* just because I want to see what happens. As it turns out, when module is coded in a Ruby file and executed, the code in module gets executed by default. This happens even if we don't include the module.

```
# module.rb

module Something
      puts "Something"
end

module Nothing
      puts "Nothing"
end
```

Output

```
Something
Nothing
```

The output of the above program *module.rb* prints out *Something* and *Nothing*. We have put two

26 Methods are another name for function

modules called *Something* which contains the code *puts "Something"* and another module Nothing which contains *puts "Nothing"*. Though I haven't included these modules in my program using *include* statement, the code under them gets executed anyway.

Calling functions without include

In the program *module_function.rb* , we have seen how to include module and call the function(s) in it. We printed a line of stars and dollars. Lets do the same in a different way. This time we wont be using the *include* keyword.

Type the program *module_function_1.rb* and execute it.

```
# module_function_1.rb

module Star
      def Star.line
            puts '*' * 20
      end
end

module Dollar
      def Dollar.line
            puts '$' * 20
      end
end

Dollar::line
Star::line
Dollar::line
```

Output

```
$$$$$$$$$$$$$$$$$$$$
********************
$$$$$$$$$$$$$$$$$$$$
```

Take a look at the following code:

```
Dollar::line
Star::line
Dollar::line
```

When we call *Dollar::line* , the *line* function in *Dollar* module gets executed. When we call *Star::line* , the *line* function in the *Star* module gets executed. Doesn't it look simple that to use *include* statement? So when you want to call a function in module just use the following syntax

<module-name>::<function-name>.

Note that in module *Star*, we have defined the function *line* as *Star.line* and not just *line*. Similarly in module *Dollar* we have defined it as *Dollar.line*.

OK, we are getting to know about modules, now lets get our hands really dirty. Type the code below (*module_function_2.rb*) and execute it.

```
# module_function_2.rb

module Star
      def Star.line
            puts '*' * 20
      end
end

module Dollar
      def Dollar.line
            puts '$' * 20
      end
end

module At
      def line
            puts '@' * 20
      end
end

include At

Dollar::line
Star::line
Dollar::line
line
```

Output

```
$$$$$$$$$$$$$$$$$$$$
********************
$$$$$$$$$$$$$$$$$$$$
@@@@@@@@@@@@@@@@@@@@
```

OK you have got some output. Take a look at the following lines

```
include At

Dollar::line
Star::line
Dollar::line
line
```

Note that we have included the module *At* at first (see highlighted code above) using the *include At* statement. While executing the *Dollar::line* statement, we get an output of twenty dollars which forms a line. While executing *Star::line* we get a output of twenty stars. Next we once again call *Dollar::line*, then comes the catch. We just call the function *line*. Since we have included *At* at first, when the statement *line* is encountered it calls the *line* method in *At* module gets called. This shows that though we have called *Dollar::line* and *Star::line* , it does not include[27] the module code in the program, instead it just executes the particular function in the module.

In *module_function _1.rb* , we have seen how we can call a function in a module say *Star::line* , where Star is the module name and line is the function name. To do so in Star module we have defined the function line as follows

```
def Star.line
      puts '*' * 20
end
```

Where instead of naming it just line, we have named it *Star.line*. Note that module_function_3.rb is similar to *module_function_1.rb*, but take a deep look into line function in Dollar module. It is not named *Dollar.line* , instead its named *Star.line* . What will happen if we mess up the code like this? Execute the program below and see.

```
# module_function_3.rb

module Star
      def Star.line
            puts '*' * 20
      end
end

module Dollar
      def Star.line
            puts '$' * 20
      end
end

module At
      def line
            puts '@' * 20
      end
end

include At
```

27 Or mixin

```
Dollar::line
Star::line
Dollar::line
line
```

Output

```
@@@@@@@@@@@@@@@@@@@@
$$$$$$$$$$$$$$$$$$$$
@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@
```

Notice that whenever we call *Dollar::line* , the *line* function in *At* module is called. Its because since we have defined it as *Star.line* in *Dollar* module, *Dollar::line* simply does not exist and hence the function *line* in the *At* module is called. Note that we have included *At* module using the statement *include At*.

We now consider another scenario where (see program *module_function_4.rb*) in the *Dollar* module we just define the function *line* as line and not as *Dollar.line*. When we call it using *Dollar::line* in the program below, we see that the *line* function in the *At* module gets called. So the moral of the story is, if you are calling *<module-name>::<function-name>* in your program, make sure that the function is named *<module-name>.<function-name>* inside the module.

```
# module_function_4.rb

module Star
      def Star.line
            puts '*' * 20
      end
end

module Dollar
      def line
            puts '$' * 20
      end
end

module At
      def line
            puts '@' * 20
      end
end

include At

Dollar::line
Star::line
Dollar::line
line
```

Output

```
@@@@@@@@@@@@@@@@@@@@
********************
@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@
```

Classes in modules

We have seen how the Ruby interpreter behaves to functions in modules. Lets now see how it behaves to classes in modules. Type the program below *module_class.rb* and execute it.

```
# module_class.rb

module Instrument
      class Ruler
            def what_u_do?
                  puts "I take measurements"
            end
      end
end

module People
      class Ruler
            def what_u_do?
                  puts "I govern this land"
            end
      end
end

r1 = People::Ruler.new
r2 = Instrument::Ruler.new

r1.what_u_do?
r2.what_u_do?
```

Output

```
I govern this land
I take measurements
```

Lets analyze the program. We have two modules. The first one is named *Instrument*, and the second one is named *People*. Both have a class named *Ruler* and both Ruler's have method named *what_u_do?* . Fine, now lets come to the program. We have a statement *r1 = People::Ruler.new* in which *r1* becomes an instance variable of *Ruler* class of *People* module. Similarly we have *r2 = Instrument::Ruler.new* in which *r2* becomes instance variable of *Ruler* class in *Instrument* module.

This can be verified by executing the following code

```
r1.what_u_do?
r2.what_u_do?
```

In which calling *r1.what_u_do?* outputs *I govern this land* and calling *r2.what_u_do?* outputs *I take measurements*.

The moral of the story is you can have same class names in different modules, to call that class just use *<module-name>::<class-name>* .

Mixins

Another use of modules is that you can mix code in modules as you wish. This one is called mixin. We have already seen about mixins but I haven't told you that it was mixin. For example lets say that you are writing some application in Ruby for Linux and Apple machine. You find that some code works only on Linux and other works only on Apple, then you can separate them as shown below. The Apple stuff goes into Apple module and Linux stuff goes into the Linux module.

Lets say that your friend uses a Linux machine and wants to run your program. All you need to do is to include Linux in your code as shown below.

```
# mixin.rb

module Linux
      # Code for linux goes here
      def function
            puts "This function contains code for Linux systems"
      end
end

module Apple
      # Code for apple goes here
      def function
            puts "This function contains code for Apple systems"
      end
end

include Linux

function
```

Output

```
This function contains code for Linux systems
```

When the method *function* is called, the method *function* in Linux module will be called. In short you have mixed in Linux code in your program and have kept out the Apple stuff.

Lets see another example of mixin. Take a look at the code *mixin_2.rb* below. Lets say that your client tells that he is badly need of a program that computes the area of circle and volume of sphere. So you develop two classes called *Circle* and *Sphere* and equip with code to find area and volume. So your client is happy. Since your client is in Milkyway galaxy, the constant *Pi*[28] is 22 divided by 7. So we have put the value of *Pi* in a module named *Constants* and have included in *Circle* and *Sphere* class using the statement *include Constants*. Type in the program below and execute it.

```
# mixin_2.rb

module Constants
      Pi = 22.0/7
end

class Circle
      include Constants
      attr_accessor :radius

      def area
            Pi * radius * radius
      end
end

class Sphere
      include Constants
      attr_accessor :radius

      def volume
            (4.0/3) * Pi * radius ** 3
      end
end

c = Circle.new
c.radius = 7
s = Sphere.new
s.radius = 7
puts "Circle Area = #{c.area}"
puts "Sphere Volume = #{s.volume}"
```

Output

28 Pi is a mathematical constant used to find area of circle and volume of sphere. Unless you are Einstien, don't bother about it.

```
Circle Area = 154.0
Sphere Volume = 1437.333333333333
```

So you get something as output. You might ask whats so great in putting a constant in a module and mixing it in a class using `include` statement. Well the above program teaches you two morals

1. You can put constants in a module
2. If you have common code[29] that can be shared between classes, you can put it into module and share it.

If you have defined the value of `Pi` separately in each class and if you happened to get a client from Andromeda galaxy where Pi is 57 divided by 18.1364, you can make change just in one place, that is in constants module and see the change reflect in many classes (`Circle` and `Sphere` classes in our case).

Thus moral is modules help us to cater customers who are beyond our own galaxy and we can truly build a galactic empire[30].

29 Common functions and constants
30 Somewhat like a Star Wars thing. So if you have a dream to become Galactic Emperor, study Ruby.

Shebang

Lets say that you have written a ruby program called *something.rb* and want to run it. To run it, in your console you must migrate to the directory in which its present and type in *ruby something.rb* to execute it. Instead of doing such a tedious procedure why can I execute the program by just typing its name like *./something.rb* as you would do to execute a executable file. Lets find out how to do it to a ruby program.

To do so we must find out where our Ruby interpreter is installed, for that we well query where is Ruby as shown

```
$ whereis ruby
ruby: /usr/bin/ruby1.8 /usr/bin/ruby /usr/lib/ruby /usr/share/man/man1/ruby.1.gz
```

From experience on my Ubuntu system I can say my Ruby interpreter is in /usr/bin folder, I can call it or execute it using the command /usr/bin/ruby. Fire up your text editor, type in the code shown below and save it as shebang.rb

```
#!/usr/bin/ruby
# shebang.rb

puts "Just put #!/usr/bin/ruby to make your Ruby programs
execute without using ruby before them"
```

Take a look at the first line thats been highlighted. The line is called shebang[31], its starts with a comment sign # followed by a exclamation mark ! Then its immediately followed by the path to the Ruby interpreter. Thats it. Your rest of the program can follow as usual.

To execute the piece of file, we must first make this as an executable. We make it as executable using the *chmod a+x file_name* as shown

```
$ chmod a+x shebang.rb
```

Once we have done it all we need to do is to type the filename as shown below

```
$./shebang.rb
Just put #!/usr/bin/ruby to make your Ruby programs
execute without using ruby before them
```

The *./* says that the file is in current working directory. Once we type the file name, due to the shebang line the operating system passes the file to Ruby interpreter which is located in */usr/bin* and the interpreter faithfully executes the program.

31 I think its inventor was banged by his girl while he was inventing this, so he may have named it “She Bang”

Date and Time

Ruby has got ways by which we can extract time and date from our computer clock. All modern day personal computers has got a thing called RTC or real time clock which is powered by a battery and would maintain the time even if the machine is turned off. Many programming languages let us access this clock and do manipulation with date and time. Lets see how to work with Ruby. Lets do this stuff in our irb rather than writing programs into a file

The first thing we do is to find whats the time now? For that just type `Time.now` in your irb, thats it. You will get the current time.

```
>> Time.now
=> Thu Feb 25 16:54:45 +0530 2010
```

`Time.now` is a synonym to `Time.new` which creates a new Time object. You can use `Time.now` or `Time.new` , both will give the same result.

```
>> Time.new
=> Thu Feb 25 16:54:48 +0530 2010
```

In the following command, we create a new time object / instance and assign it to a variable `t`

```
>> t = Time.new
=> Thu Feb 25 16:55:02 +0530 2010
```

Now having assigned, how to view the value of `t`? We will use the inspect method. So by typing `t.inspect` , we can inspect whats in `t`.

```
>> t.inspect
=> "Thu Feb 25 16:55:02 +0530 2010"
```

`t.inspect` converts the time object to a string and displays to us. The same thing can be done by `to_s` (to string) function as shown below

```
>> t.to_s
=> "Thu Feb 25 16:55:02 +0530 2010"
```

`t.year` retrieves the year in time object

```
>> t.year
=> 2010
```

`t.month` retrieves the month in time object

```
>> t.month
=> 2
```

`t.day` retrieves the day (in that month) in time object

```
>> t.day
=> 25
```

`t.wday` retrieves the day's number. Here 0 means Sunday, 1 → Monday, 2 → Tuesday and so on till 6 means Saturday

```
>> t.wday
=> 4
```

In above code snippet, the day was Thursday

`t.yday` retrieves the day in that year. For example 1st of February is the 32nd day in the year

```
>> t.yday
=> 56
```

`t.hour` retrieves the hour in the time object. The hour is 24 hour format

```
>> t.hour
=> 16
```

`t.min` retrieves the minutes value in time object

```
>> t.min
=> 55
```

`t.sec` retrieves the seconds in time object

```
>> t.sec
=> 2
```

`t.usec` retrieves microseconds in the time object. This will be useful if you are commissioned to write a stopwatch application for Olympics.

```
>> t.usec
=> 357606
```

`t.zone` retrieves the zone. I am in India, we follow Indian Standard Time here, its spelled IST for short

```
>> t.zone
=> "IST"
```

There is a thing called UTC or Universal Time Coordinate. Its the time thats at longitude 0^{o}. The `t.utc_offset` displays the number of seconds your time is far away from the time at UTC.

```
>> t.utc_offset
=> 19800
```

From the above example, I came to know that a person living at Greenwich will see sunrise after 19800 seconds after I have seen.

DST means daylight saving time. I don't know what it is. If your timezone has a daylight saving, this function returns true, else false

```
>> t.isdst
=> false
```

If your timezone is UTC, the `t.utc` returns true or returns false

```
>> t.utc?
=> false
```

If you want to get the local time just call the local time function as shown. We want t to hold local time value in this case

```
>> t.localtime
=> Thu Feb 25 16:55:02 +0530 2010
```

In same way as local time, the `gmtime` function gets the Greenwich Meridian Time.

```
>> t.gmtime
=> Thu Feb 25 11:25:02 UTC 2010
```

The `getlocal` function is the alias of local time

```
>> t.getlocal
=> Thu Feb 25 16:55:02 +0530 2010
```

The `getutc` function is alias of `gmtime`. Actually `gmtime` is alias of `getutc`

```
>> t.getutc
=> Thu Feb 25 11:25:02 UTC 2010
```

The `ctime` function formats the time to somewhat human readable form.

```
>> t.ctime
=> "Thu Feb 25 11:25:02 2010"
```

Lets say we want to subtract some seconds from the time value `t`, we can do it as shown. Below we subract 86400 seconds (1 day) from our time value

```
>> t - 86400
=> Wed Feb 24 11:25:02 UTC 2010
```

Days between two days

Lets now write a code snippet that finds the number of days between February 25th 2010 to May 1st 2010, first we declare a variable a and assign it with the day February 25th 2010 as shown

```
>> a = Time.local 2010, 2, 25
=> Thu Feb 25 00:00:00 +0530 2010
```

Notice we use a function called `local` in Time class or object, we can assign a date to it. As we could see in the output, we get to know that variable `a` now has the value of February 25th . In similar fashion we create a variable b and assign it with date 1st of May 2010

```
>> b = Time.local 2010, 5, 1
=> Sat May 01 00:00:00 +0530 2010
```

All we do now is subtract a from b

```
>> b-a
```

```
=> 5616000.0
```

This gives number of seconds between *a* and *b*. We divide the result by 86400 (thats how many seconds that are in a day)

```
>> days = _ / 86400
=> 65.0
```

We get a result as 65. Cool!

How many days have you lived?

Lets now see a program that takes in your birthday and prints out how many days have you lived. Type in the program in text editor and execute it

```
#!/usr/bin/ruby
# how_many_days.rb

print "Enter birthday (YYYY-MM-DD):"
bday = gets.chop
year, month, day = bday.split('-')
# puts " #{year}, #{month}, #{day}"
seconds =  Time.now - Time.local(year, month, day)
days = (seconds / 86400).round
puts "You have lived for #{days} days"
```

Here is the result

```
Enter birthday (YYYY-MM-DD):2000-5-23
You have lived for 3566 days
```

Well this may vary on when you execute this program. Lets now analyze it. In the first line

```
print "Enter birthday (YYYY-MM-DD):"
```

We ask the user to enter his or her birthday, once done we perform a trick here. We asked the user to enter it in YYYY-MM-DD format, in the statement

```
bday = gets.chop
```

We get the date and store it in a variable called *bday*. The *gets.chop* gets the birth day and chops off the enter sign we enter with it. So *bday* now holds the string value of birthday you entered. In the next statement

```
year, month, day = bday.split('-')
```

we have a multiple assignment in which I have three variables year, month and day. I am splitting the birthday string and assigning it. What really happens is this, if we enter a date like 1994-6-24 it gets split by – and becomes an array which is shown in code snippet below executed in irb

```
>> "1994-6-24".split '-'
=> ["1994", "6", "24"]
```

Lets assign this array to variables a, b, c simultaneously as shown

```
>> a, b, c = _
```

```
=> ["1994", "6", "24"]
```

If you remember _ (underscore) means the last obtained result in irb. So having assigned it we now check values of a, b and c which we get as shown....

```
>> a
=> "1994"
>> b
=> "6"
>> c
=> "24"
```

Isn't ruby spectacular? In similar fashion in

```
year, month, day = bday.split('-')
```

The *year* variable gets the year part, the *month* gets the month and *day* gets the day. OK having obtained the parameters of a particular day we can proceed. Examine the following statement

```
seconds =  Time.now - Time.local(year, month, day)
```

See the right hand side of the equal to sign, first we have *Time.now* which gets the current time, from it we are subtracting a time object thats been created using *Time.local*. *Time.local* can be used to create a time object thats fixed at any instance, the instance can be past, present or future. We pass the year month and day to it to create a Time object. What happens when we subtract these both, we get the difference in seconds which gets stored in variable called *seconds* at the left hand side of the equation.

All we need to do now is to convert the second to days which is done by the following statement

```
days = (seconds / 86400).round
```

Here we divide seconds by 86400 which converts them to days. We might be getting some value like 378.567 days, to get rid of the .567 we round it off using the *round* function, so *(seconds / 86400).round* returns a neat rounded value which can be read by humans quiet easily. We store the value in a variable called days. Finally we print the fact that we have lived for so many long days using the following statement

```
puts "You have lived for #{days} days"
```

Well thats it.

I would like to tell one thing I found out with *Time.local* function, its not that we must pass only numbers to it as shown

```
>> Time.local "2010", "5", "1"
=> Sat May 01 00:00:00 +0530 2010
```

We can pass a bit human friendly values as shown below. Instead of putting 5 for month, we use May.

```
>>Time.local "2010", "may", "1"
=> Sat May 01 00:00:00 +0530 2010
```

Some times Ruby language looks as easy as talking English

Files

Until now you have stored data in variables in your program. Variable data gets lost when the program stops executing or when the computer is switched off. If you want a persistent storage you must store it in files. When you store data in files, it stays there even if the computer is shut down, and you can get the data back when its turned on. This very book if you are reading it on computer, kindle or electronic reader is a file thats stored permanently on your computer. In this chapter we will see how we can create, manipulate and delete files using ruby program.

Storing output into files

Lets create a familiar Ruby program. Type the program below into a text editor

```
#!/usr/bin/ruby
# write_file.rb

puts "Hello World!"
puts "Ruby can write into files"
```

While executing it, give command as shown

```
$ ruby write_file.rb > something.txt
```

Now goto the working directory in which the program was and you will see a file named `something.txt` . Open it and this is what you will see in it

```
Hello World!
Ruby can write into files
```

Well, this time its somewhat like a cheat. We haven't written into a file in our program, instead we have instructed the ruby interpreter to take the output that *`write_file.rb`* generates and put it in a file called `something.txt`. To do so we make use of `>` (greater than sign).

Taking file as input

In the last example we wrote our output to a file. Now lets take a file as input and lets process it. Write the code below in text editor and save it as line_count.rb.

```
#!/usr/bin/ruby
# line_count.rb
```

```
puts "The file has #{readlines.length} line(s)"
```

To execute it, we will give command as shown

```
$ ruby line_count.rb < something.txt
```

If you have guessed it right, we given *something.txt* as input to the program. We use the < (less than)sign to indicate to the program that we are giving a file as input.

The program when executed provides the following result

```
The file has 2 line(s)
```

Lets analyze the program so we know what happens. See the code that been highlighted in the program above. The *readlines* command takes in the file and reads all the lines and stores it in an array, each element of the array has a line. All we have to do is to get the length of an array which we can get by using the *length* function. So *readlines.length* gives the length as output which we embed it into a string hence we finish it off by writing the following statement

```
puts "The file has #{readlines.length} line(s)"
```

File copy – a kind of

Well, here is file copy program which might make some pundits argue weather if this program is a true file copy program or not. Type the program below into a text editor

```
#!/usr/bin/ruby
# file_copy.rb

puts readlines.join
```

And run it like this

```
$ ruby file_copy.rb < something.txt > everything.txt
```

Output

everything.txt has got everything that *something.txt* has got. Just open it and see for yourself.

The trick is in the commandline. Here we pass *something.txt* to the program *file_copy.rb* , now the program takes in *something.txt* and it reads the lines when it encounters the *readlines* command. The lines that are readed from *something.txt* are stored in the form of an array. All we now have to do is to join the lines stored in array using *join* command, so we do it by adding *.join* to *readlines*, hence we get

```
readlines.join
```

Now we will print out the result, we do this by using a puts command, hence our program takes the following incarnation

```
puts readlines.join
```

While running the program we tell it to take input from *something.txt* and write the generated result to *everything.txt* by giving the following command to Ruby interpreter

```
$ ruby file_copy.rb < something.txt > everything.txt
```

So we get the effect of copying without really writing a program for file copy.

Displaying a file

Lets now write a program that displays the content of a file. To do so we read all lines in a file, store it in an array. Next we take each and every element of an array and print it out. Type the program below and

```
#!/usr/bin/ruby
# display_file.rb

readlines.each do |line|
      puts line
end
```

Execute it by typing the following

```
$ ruby display_file.rb < something.txt
```

This is what you will get as output

```
Hello World!
Ruby can write into files
```

So what we have done in this program.? Look at the highlighted code block, when ruby encounters *readlines* , it reads from the file passed to the program, extracts the lines and stores it in an array. With *.each* operator we extract a single line at a time and store it into a variable called *line* inside the *do end* block. We print out this *line* in the code block using *puts*. The end statement put an end to the code block says all is over.

Lets see another program. In this program we use a more cleaner approach. Type the program into a text editor and execute it

```
#!/usr/bin/ruby
# display_file_1.rb

puts File.open("something.txt").readlines
```

Output

```
Hello World!
Ruby can write into files
```

Look at the single line in the program. We have a *puts* statement, that prints almost what ever is thrown at it. Here is the new thing thats been introduced. Look at the highlighted code, we have a thing called *File.open("something.txt")* , the *File.open* opens a file, but what file? We must

give the name of the file to it. As a file name we pass *something.txt* in double quotes[32]. The *File.open* opens it, the *.readlines* attached to it reads lines and stores it in an array. We throw the array to *puts* which prints it out. Thats it!

Reading file line by line

Till the last section we have seen how to read a file in a go and pump its data out to the console. In this example we will see how to read a file line by line. Type in the example code given below and execute it

```
#!/usr/bin/ruby
# read_file_1.rb

File.open("something.txt").each { |line| puts line }
```

Output

```
Hello World!
Ruby can write into files
```

The output looks as shown above. Look at the highlighted code In the code we open a file named *something.txt* using a *File.open* command which opens the file and stores the lines as array elements. All we need to do now is to extract each element in an array and print it out on our console which is accomplished by the line highlighted above.

Instead of using *File.open* , one could use *File.new* to open a file. All will have the same result. A program using *File.new* has been written and is shown below, execute it and you will get the same result.

```
#!/usr/bin/ruby
# read_file_2.rb

File.new("something.txt").each { |line| puts line }
```

Output

```
Hello World!
Ruby can write into files
```

Open and new – the difference

Seen from previous examples one might think that there isn't much difference between File.open and File.new, infact there is a difference. Consider the program below, type it and execute it

```
#!/usr/bin/ruby
# file_open.rb
```

32 A name is a string right?

```
File.open("something.txt") do |f|
  puts f.gets
end
```

Output

```
Hello World!
```

The program above prints out the content present in something.txt, the same thing is done by file_new.rb as shown below

```
#!/usr/bin/ruby
# file_new.rb

f = File.new("something.txt", "r")
  puts f.gets
f.close
```

Output

```
Hello World!
```

OK so whats the difference? *File.new* returns a new file object or handle that can be store into a variable. In the above program we store the file object into variable *f*. We can use this variable any where in the program to access and manipulate the file. Having done all needed with the file using the variable *f*, we finally close the file using *f.close*.

Lets write a program named *file_open_error.rb* as shown below

```
#!/usr/bin/ruby
# file_new.rb

File.open("something.txt") do |f|
  puts f.gets
end
puts "Reading file after File.open block is closed:"
puts f.gets # This should throw an error
```

Output

```
Hello World!
Reading file after File.open block is closed:
file_open_error.rb:8: undefined local variable or method `f' for main:Object
(NameError)
```

See the highlighted code, we try to read the file content after we close the code block and it throws an error, this is because *File.open* loads into file handle into variable *f* inside the do end code block, after the block is closed you have no way to access the file.

Though the difference is minor, there is still a difference.

Defining our own line endings

Till now reading line by line means that the Ruby program when given a file name searches for it, loads it then it scans the file, when it encounters a line ending character '\n'[33] on the Linux system (its \r\n on windows) it recognizes the line has ended and hence packs the characters before it into an array element. What if we want to define our own line ending character? In English language full stop is considered as a line ending character. Why can't we say to the Ruby interpreter to mark end of the line at a full stop character? To do so lets create a simple text file named *line_endings.txt* and put the following text in it

```
This is first line. This is second line. This
is the third. And fourth comes after third.
```

Lets write a Ruby program shown below in text editor, save it as *line_endings.rb*

```
#!/usr/bin/ruby
# line_endings.rb

File.open("line_endings.txt").each('.') do |line|
      puts line
end
```

When executed the program prints out the following output

```
This is first line.
 This is second line.
 This
is the third.
 And fourth comes after third.
```

See carefully *line_endings.txt* . This is first line : *This is first line.*

and This is second line : *This is second line.*

Both are on the same line in *line_endings.txt* but it gets printed out as two different lines when the program is executed. This is because the statement *File.open("line_endings.txt")* loads the entire content of the file into the memory, the *.each('.')* splits the content at every dot or full stop character ('.'), and puts the each chunk of split text into an array element. So the real hero here is the each function. In a similar way you can have any character that can define a line ending.

If you are writing a C compiler using Ruby, you might use the semicolon character (;) as your line ending.

33 The \n character will not be shown to the user and hence you wont be able to see it when you open the file with a text editor.

Reading byte by byte

Sometimes you want to read a file byte[34] by byte instead of reading plain English in it. Why on earth we read a file byte by byte? Well, not all files have text in it. Files such as music files, videos and so on have raw data which only some programs can understand. If you are writing a music or video player or image viewer, you need to read the raw data and do something with it. So to read and display bytes of data we use *each_byte* function. Take a look at the code below. Type it and execute it

```
#!/usr/bin/ruby
# byte_by_byte.rb

File.open("something.txt").each_byte { |byte| puts byte }
```

When executed this is how the output will look like

```
72
101
108
108
111
32
87
111
.
.
some stuff is removed to save pages printed
.
.
105
108
101
115
10
```

In the above program we open the file named *something.txt* using *File.open* , all the contents gets loaded, now we access the content byte by byte using the *each_byte* function, we capture the bytes in variable called byte and print it out. Notice that in this program we have used curly brackets { and }, these can be used instead of *do* and *end* . I prefer *do* and *end* as they look more friendly.

Reading single character at a time

The program below reads character by character and prints it. We use a function called *each_char* which works with Ruby 1.9.x version. Currently I have not installed it, so I am not giving away the output of this program.

34 To know what byte is goto http://wikipedia.org/byte

```
#!/usr/bin/ruby
# char_by_char.rb

# To get this program to work, you must
# have ruby 1.9

File.open("something.txt").each_char { |a| puts a }
```

Output

I don't have ruby 1.9 installed on my machine.

Renaming files

Renaming a file is extremely easy in Ruby, all you have to do is to call the *rename* function in file class. The first argument will be the name of the file that needs to be renamed, the second one will be the new name. Its so simple you can try it out on the irb. Take a look at the source code of program *rename.rb* given below. In it we rename a file called *noname.txt* to *somename.txt*. Before you run the program place a file called *noname.txt* on the working directory.

```
#!/usr/bin/ruby
# rename.rb

File.rename("noname.txt", "somename.txt")
```

Output

The file noname.txt was renamed to somename.txt

Finding out position in a file

You might sometime need to find out your position within a file. To do so you can use the method *pos*. Lets see an example that explains us how to find our position in a file. Type and execute *fie_position.rb*

```
#!/usr/bin/ruby
# file_position.rb

f = File.open "god.txt"
puts "At the beginning f.pos = #{f.pos}"
f.gets
puts "After reading first line f.pos = #{f.pos}"
f.gets
puts "After reading second line f.pos = #{f.pos}"
```

Output

```
At the beginning f.pos = 0
After reading first line f.pos = 43
After reading second line f.pos = 69
```

Lets now walkthru the code and see how it works. First we open a file named *god.txt* in the line *f = File.open "god.txt"* next we check out whats the position using the statement *puts "At the*

*beginning f.pos = **#{f.pos}"*** , note the *f.pos*, the *pos* method is used to get the position that we are in while we read or write a file. Initially when we open a file the position will be at zero and so we get the following output

```
At the beginning f.pos = 0
```

In the next line we read the first line of file using f.gets , since we have read the file like the reading pointers position should have changed[35] , so when we print *f.pos* it must display some other number than zero. So the statement *puts "After reading first line f.pos = #{f.pos}"* produces the following result

```
After reading first line f.pos = 43
```

Just for the sake of educating more we read the second line using another *f.gets* now we print the new file position, now we find that the pointer points to position *69*.

If you are wondering what *god.txt* has, here is it:

```
All things exists because it was created.
Then the creator exists.
Did man ever think how the creator exist?
If such a mighty creator can exist without creation,
then why can't this simple universe exist without
a creator?
```

In the coming example we will see how to change our position within a file. Type the example below (*file_changing_position.rb*) and execute it

```
#!/usr/bin/ruby
# file_changing_position.rb

f = File.open "god.txt"
puts "Reading file with f.pos = 0"
puts f.gets
puts "_"*40
f.pos = 12
puts "Reading file with f.pos = #{f.pos}"
puts f.gets
puts "Now f.pos = #{f.pos}"
```

Output

```
Reading file with f.pos = 0
All things exists because it was created.
_____________________________________________
Reading file with f.pos = 12
xists because it was created.
Now f.pos = 43
```

Read the program carefully and notice the output. First we open the file *god.txt* and the variable *f* has its handle. Next in line

```
puts f.gets
```

35 In this case, should be at the end of file

We are reading with file with *f.pos* at zero, that is we are reading from the start of file. As you can see the output for the first *puts f.gets* we get the entire line *All things exists because it was created.* gets printed. Notice the next line carefully, we now change our position within file to position 12 using the statement *f.pos = 12*, this means that our pointer is 12 bytes from the start. Now in the second *puts f.gets* , we get the output as *xists because it was created.* This shows us that we are able to change our position within a file in a successful way.

Some minds could think that there could be a possibility of negative file position where say if you want to read the last 20 bytes of file you can assign *f.pos = -20* and when giving *f.gets* it would get printed. Well, thats not possible with Ruby. If you want try out the example (*file_negative_position.rb*) and see weather it gives a proper result.

```
#!/usr/bin/ruby
# file_negative_position.rb

# this example wont work

f = File.open "god.txt"
f.pos = -20
puts "Reading file with f.pos = #{f.pos}"
puts f.gets
```

Writing into files

Till now we have seen how to read from files, we will now see how to write content into files. To learn how to write into files type the below example (*write_file_1.rb*) into the text editor and execute it

```
#!/usr/bin/ruby
# write_file_1.rb

File.open "god.txt", "w" do |f|
      some_txt = <<END_OF_TXT
All things exists because it was created.
Then the creator exists.
Did man ever think how the creator exist?
If such a mighty creator can exist without creation,
then why can't this simple universe exist without
a creator?
END_OF_TXT

      f.puts some_txt
end
```

After execution open the file *god.txt* and this is what you will see in it

```
All things exists because it was created.
Then the creator exists.
```

```
Did man ever think how the creator exist?
If such a mighty creator can exist without creation,
then why can't this simple universe exist without
a creator?
```

Lets walk thru the program and see how it works. First in the statement `File.open "god.txt", "w"` , we open a file named `god.txt` for writing. We indicate that we are opening the file for writing by passing `"w"` as second argument. This second argument is called as a flag. Given below are list of flags that can be used for file operations.

Flag	What it says
r	The file is opened in read only mode. The file pointer is placed at the start of file.
r+	In r+ mode both reading and writing is allowed. The file pointer is placed at the start of the file
w	This means write only. If the file does not exist, a new file is created and data is written into it. If the file exists the previous content is replaced by new content
w+	In this mode both reading and writing is allowed. If the file does not exist, a new file is created. If it exists the old content is lost and new one s written.
a	This flag opens the file in append mode. Append mode is a special form of write mode in which the new content added is placed the end of old content[36], by this way previous information isn't lost.
a+	Both reading and writing is allowed (i.e append mode plus reading and writing). Any newly added data is placed at the end of the file.
b	Binary file mode. In this mode files that have data other than text is read. For example opening a music or video file.

Having opened a file in write mode we now have opened a `do end` block within which we capture the file handle in variable `f`. All we need to do is to write a string to the file.

We create a string using the following code

```
some_txt = <<END_OF_TXT
All things exists because it was created.
Then the creator exists.
Did man ever think how the creator exist?
If such a mighty creator can exist without creation,
then why can't this simple universe exist without
a creator?
END_OF_TXT
```

Now `some_txt` has got a string which we need to write it into the file. To write it into the file we use the following statement

```
f.puts some_txt
```

`gets` gets the file content, `puts` writes something into the file, so as an argument to the puts function we pass `some_txt` , the content held in it gets written into the file. The program reaches the

36 The file pointer is placed at the end of the file

end, the file is closed and that's it. When you open *god.txt* you can see what's written in it.

Appending content into files

Till now we have seen how to read from files and write content in it. Now lets see how to append content in it. While appending content into files, the old content stays on the new content is added at the bottom of the page.

To understand how this works type the program *file_append.rb* and execute it.

```
#!/usr/bin/ruby
# file_append.rb

puts "Enter text to append into file: "
text = gets
f = File.new("log_file.txt", "a")
f.puts "\n"+Time.now.to_s+"\n"+text
```

When the program prompts you to enter some thing, type some thing like *It will be great if dinosaurs were still around* and press enter. Run this program a few times, type something, after you got bored from few run's open *log_file.txt* and see what it contains. When I opened mine, this is what I got:

```
Sat Mar 27 16:20:24 +0530 2010
This is my first log

Sat Mar 27 16:21:10 +0530 2010
This is my second log

Sat Mar 27 16:21:36 +0530 2010
This is my third log. Now I'm getting bored.
```

See how neatly your entries have been recorded along with time stamp. To understand how the program lets walk thru it.

The first line *puts "Enter text to append into file: "* , prints out *Enter text to append into file:* and the control goes on to the next line *text = gets* at which stage the program waits for you to enter something and press enter. When you do press enter, what you entered gets stored in variable *text*.

The next line *f = File.new("log_file.txt", "a")* is the crucial one and highlight of our program. In this line we open a file called *log_file.txt* in append mode. Notice that we pass *"a"* as the second argument to *File.new* which tells that we are opening it in append mode. In this mode the content that was previously stored in the file is not erased and/or over written, instead

whats new being added is written at the end of the page. (If you are reading PDF version) You can a list of file flags by clicking here.

Once having opened in append mode, all we need to do is to put content stored in variable `text` into the file. Since the file handle is stored in variable `f`, we could have completed the program by writing `f.puts text` , but I wished it would be better if we logged our data with time stamps, and I have left line breaks before and after each log so that it will be nice to read, so I have written the code `f.puts "\n"+Time.now.to_s+"\n"+text` .

Thats it, the content we have written at the program prompt and along with the time stamp gets stored into the file. At the end of the program it would have been nice if we had closed the file using `f.close` , I haven't done it in this program, but it works.

Storing objects into files

Till now we have seen to read, write and append into files, whats we stored and read were pieces of text. Now we will see how to store objects or instance of classes into files.

Pstore

Pstore is a binary file format into which you can store almost anything. In the coming example we are going to store few objects that belongs to the square class. First we will be writing a class for square and put it into a file called `square_class.rb` . If you feel lazy copy the content and below and put it into the file, if you are a active guy/gal type it all by yourself, finally you will end up with the same thing.

```
# square_class.rb

class Square
      attr_accessor :side_length

      def initialize side_length = 0
            @side_length = side_length
      end

      def area
            @side_length * @side_length
      end

      def perimeter
            4 * @side_length
      end
end
```

Once the square class is ready, we will use it in two different places. The first one is coming right

now. We create a program called *pstore_write.rb* , type the content given below in it

```
#!/usr/bin/ruby
# pstore_write.rb

require 'square_class.rb'

s1 = Square.new
s1.side_length = 4
s2 = Square.new
s2.side_length = 7

require 'pstore'
store = PStore.new('my_squares')
store.transaction do
      store[:square] ||= Array.new
      store[:square] << s1
      store[:square] << s2
end
```

We will walk thru the program now. The first line *require 'square_class.rb'* includes the code of the square class into the program, by doing so we can write code as though the square class code is typed into the same file, this reduces lot of typing and makes the code look neat.

In the next four lines shown below, we declare two squares *s1* and *s2*, we assign *s1*'s *side_length* to be 4 units and that of *s2* to be 7. In the next line we include the code needed to read and write the pstore file format. We don't need to write that code as its already written for us, all we need to do is to type *require 'pstore'* and that will include the code.

Next we create pstore file using the command *store = Pstore.new('my_squares')* . This creates a pstore file called *my_squares* and passes on the file handle to the variable named *store*, with this variable *store* we can read, manipulate the file *my_squares*. To start writing into the file we need to start a transaction which is accomplished by the following block of code

```
store.transaction do

end
```

Now we can do transactions with the pstore file within the *do* and *end* block. Within the block we add the code thats highlighted below

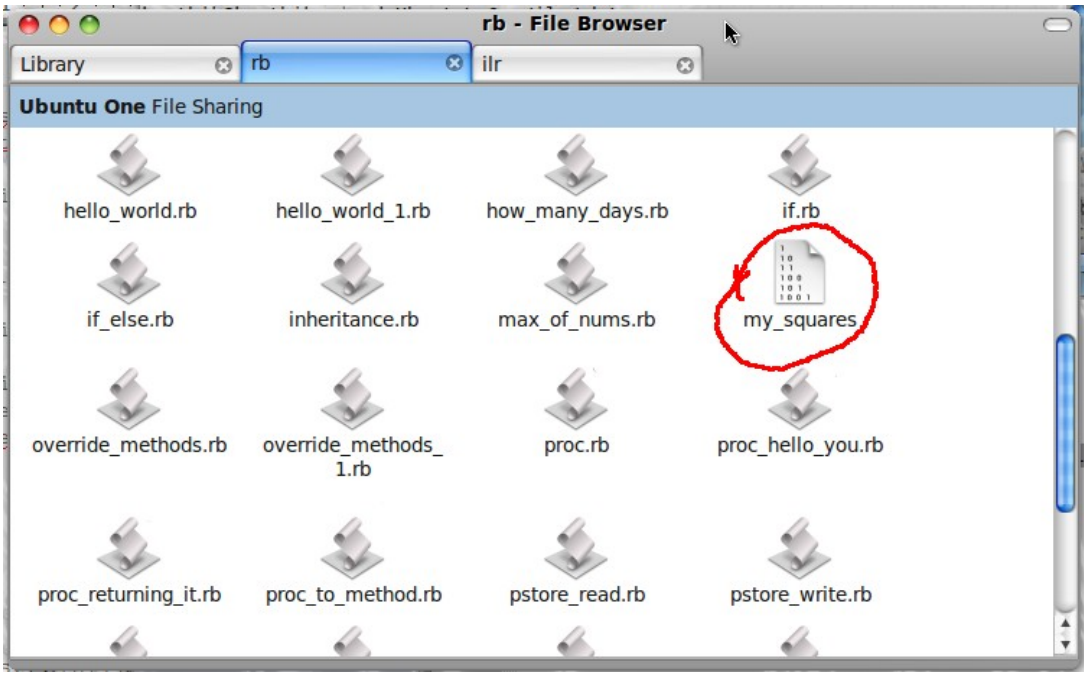
```
store.transaction do
      store[:square] ||= Array.new
      store[:square] << s1
      store[:square] << s2
end
```

The first line creates a array named *store[:square]*, the *||=* means that if already a variable named *store[:square]* exists then there is no need to create that variable as its already there. If such a variable doesn't exist, the we need to create it. After creating an array we we add *square*

objects / instance variables *s1* and *s2* into them using the following lines

```
store[:square] << s1
store[:square] << s2
```

Once done we close the transaction using the end command. Just view your working directory, you will be able to see a file named *my_squares* in it as shown in image below:



So now we have successfully written into the pstore file named *my_square* . All we need to do is read it and confirm what we have done is right. To read the data written into it we will write a program *pstore_read.rb* .

Create a file named *pstore_read.rb* and store the program written below in it, execute and watch the output.

```
#!/usr/bin/ruby
# pstore_read.rb

require 'square_class.rb'
require 'pstore'

store = PStore.new('my_squares')
squares = []
store.transaction do
  squares = store[:square]
end

squares.each do |square|
      puts "Area = #{square.area}"
      puts "Perimeter = #{square.perimeter}"
      puts "================================"
end
```

Output

```
Area = 16
Perimeter = 16
==============================
Area = 49
Perimeter = 28
==============================
```

As you see the area and perimeter of the two squares are printed. If you feel I am tricking you check for our self with a calculator. Well to understand what happens in *pstore_write.rb* lets walkthru the code. In the first two lines

```
require 'square_class.rb'
require 'pstore'
```

we include the code in *square_class.rb* and code for reading and writing pstore files into our program. Just like the previous example we open the pstore file *my_squares* and store the file handle into the variable named *store* in the following line

```
store = PStore.new('my_squares')
```

Now we create a array named squares in the following line

```
squares = []
```

With the *store* variable (which is the *my_squares* handle) we open a transaction as shown

```
store.transaction do
  squares = store[:square]
end
```

In the transaction as shown in the highlighted code above we transfer the objects in variable *store[:squares]* to the declared variable squares, so by this time the variable square must contain the content of two square objects which we defines in previous example *pstore_write.rb*

Once we have taken out the values we can close the transaction using the *end* key word.

In the following code

```
squares.each do |square|
      puts "Area = #{square.area}"
      puts "Perimeter = #{square.perimeter}"
      puts "=============================="
end
```

we take each each object in array *squares* and load it into variable called *square* and we print out the perimeter and area.

YAML

YAML stands for YAML ain't XML. YAML is a markup language in which we can store something like data contained in Ruby objects. Lets write a program in which we store the data of the square

objects into YAML and retrieve it. Note that in this program we are not saving the output YAML data into a file, why so? Simply because I am lazy enough. Type the code *yaml_write.rb* into text editor and execute it

```
#!/usr/bin/ruby
# yaml_write.rb

require 'yaml'
require 'square_class'

s = Square.new 17
s1 = Square.new 34
squares = [s, s1]
puts YAML::dump squares
```

When executed, the program will produce the following output

```
---
- !ruby/object:Square
  side_length: 17
- !ruby/object:Square
  side_length: 34
```

Lets now walkthru the program. The first two lines

```
require 'yaml'
require 'square_class'
```

Imports the code needed to read and write into YAML files. The next one loads the code in the *square_calss.rb* so that you can program with square objects.

In the following lines

```
s = Square.new 17
s1 = Square.new 34
```

We declare two Square objects. One has edge or side length of 17 units and other has side length of 34 units. In the next line

```
squares = [s, s1]
```

We pack the objects s and s1 into an array called squares. In the following line

```
puts YAML::dump squares
```

we dump the formed array into YAML and print it onto the screen using *puts* statement.

Copy the stuff that comes in as output. It will be used to write the next program *yaml_read.rb* , type the code *yaml_read.rb* thats shown below into the text editor and execute it

```
# yaml_read.rb

require 'yaml'
require 'square_class'

yaml = <<END
---
- !ruby/object:Square
```

```
  side_length: 17
- !ruby/object:Square
  side_length: 34
END

squares = YAML::load(yaml)
squares.each do |square|
puts "Area = #{square.area}"
puts "Perimeter = #{square.perimeter}"
puts "=============================="
end
```

Look at the output

```
Area = 289
Perimeter = 68
==============================
Area = 1156
Perimeter = 136
==============================
```

The first set of area and perimeter thats been displayed is of Square `s` and second set is of Square `s1`. Lets walkthru the code understand what is happening. As usual these lines:

```
require 'yaml'
require 'square_class'
```

imports the code needed for YAML and second one imports code in `square_class.rb` which enables us to deal with `Square` objects. Next we have a multi line string `yaml`

```
yaml = <<END
---
- !ruby/object:Square
  side_length: 17
- !ruby/object:Square
  side_length: 34
END
```

The content of `yaml` is enclosed between `<<END` and `END` , note that the content of `yaml` is the output of the previous program. Concentrate on this line

```
squares = YAML::load(yaml)
```

Its here all magic happens. Here the Ruby magically finds out from the YAML that we are loading data stored in an array, this array consists of two objects of class `Square` and first one has side length 17 units and another of 34 units. So the `YAML::load` phrases it into array of Square's and stores it into variable `squares`.

In the following code:

```
squares.each do |square|
puts "Area = #{square.area}"
puts "Perimeter = #{square.perimeter}"
puts "=============================="
end
```

We load each element of array into a variable square and print its area and perimeter. If I have got

mood enough I will be putting examples of storing YAML into files and reading it back, but for now forgive my laziness.

Proc and Blocks

If you have know some programming languages, you might have heard about closures. Proc and Blocks are similar kind of thing. You can take a piece of code, stick it in between a *do end*, assign it to a variable. This variable contains the piece of code and can be manipulated like objects, passed around and blah blah.

Proc is like a function, but its an object. Lets see an example to know what an Proc is. Type in the program *proc.rb* into the text editor and execute it.

```
#!/usr/bin/ruby
# proc.rb

say_hello = Proc.new do
      puts "Hello world!"
end

say_hello.call
say_hello.call
```

This is how the output will look like

```
Hello world!
Hello world!
```

Lets now walkthru the code to understand it. Take a look at the following lines

```
say_hello = Proc.new do
      puts "Hello world!"
end
```

In this case you are taking a single Ruby statement *puts "Hello World!"* and putting it between an *do* and *end*. You are making this code a Proc by appending *Proc.new* before the *do* (the start of the block). You are assigning the Proc object to a variable named *say_hello*. Now *say_hello* can be thought as something that contains a piece of program.

Now how to call or execute the code? When we need to call the piece of Proc named *say_hello* we write the following command

```
say_hello.call
```

In the *proc.rb* we call *say_hello* twice and hence we get two *Hello World!* as output.

Passing parameters

Like functions you can pass parameters to a Proc. To see how it works, type the program *proc_hello_you.rb* and execute it.

```
#!/usr/bin/ruby
# proc_hello_you.rb

hello_you = Proc.new do |name|
      puts "Hello #{name}"
end

hello_you.call "Peter"
hello_you.call "Quater"
```

When executed the program gives the following output.

```
Hello Peter
Hello Quater
```

Take a look at the bold / highlighted / darkened code in above program. Notice that we are capturing some thing after the *do* keyword, by giving *do |name|* we are putting something thats been passed to the Proc block into the variable *name*. This variable can now be used anywhere in the Proc block to do something.

In *puts "Hello #{name}"* , we print hello followed by the passed parameter (*name*). So we have written a Proc that can accept some thing passed to it. How do we call and pass something to it? Well, notice the statements after the *end*, look at the first one, it says

```
hello_you.call "Peter"
```

hello_you.call calls the Proc code to be executed. To the Proc we pass the string *"Peter",* this string gets copied in to the variable *name* in the Proc and hence we get the output *Hello Peter*.

In a similar way when Ruby interpreter comes across *hello_you.call "Quarter"* , it prints *Hello Quater*.

Passing Proc to methods

Just like any object, we can pass Proc to methods. Take a look at the code below (*proc_to_method.rb*). Study it, code it and execute it

```
#!/usr/bin/ruby
# proc_to_method.rb

# An exampleof passing a proc to method

def execute_proc some_proc
      some_proc.call
```

```
end

say_hello = Proc.new do
      puts "Hello world!"
end
```

execute_proc say_hello

This is how the output will look like.

```
Hello world!
```

Lets now analyze the code of *execute_proc* function, its code is as follows

```
def execute_proc some_proc
      some_proc.call
end
```

We take in one argument called *some_proc* which we assume it as Proc. We then execute it by using its own *call* method, i.e. in function we just call *some_proc.call* for the passed Proc to be executed. If you look at next few lines we create a Proc called *say_hello*

```
say_hello = Proc.new do
      puts "Hello world!"
end
```

All we do in *say_hello* Proc is to print *Hello world!* And thats it. Now we call the method *execute_proc* and pass *say_hello* in the following piece of code

```
execute_proc say_hello
```

Having passed the Proc, it gets copied to *some_proc* argument and when *some_proc* is executed, it prints *Hello world!* faithfully.

Returning Proc from function

I had written earlier that Proc can be treated just like objects. In fact Proc is an object. We can pass it to functions, which we have seen in the previous subsection, now we can see an example in which Proc is returned from a function. Take a good look at code given below (*proc_returning_it.rb*).

```
#!/usr/bin/ruby
# proc_returning_it.rb

# Function that returns a proc
def return_proc
      Proc.new do |name|
            puts "The length of your name is #{name.length}"
      end
end

name_length = return_proc
name_length.call "A.K.Karthikeyan"
```

When executed, the program throws the following output

```
The length of your name is 15
```

Look at the function *return_proc* . In it we create a new Proc which accepts a parameter called *name*. Assuming that *name* is a string, we simply print the length of it. Now consider the code that comes after this function which is as follows:

```
name_length = return_proc
name_length.call "A.K.Karthikeyan"
```

In the first line *name_length = return_proc* , the *name_length* gets assigned with what ever the *return_proc* method returns. In this case since *Proc.new* is the last statement / block in the *return_proc* method, it returns a new Proc which gets assigned to *name_proc*. So now *name_proc* can accept a parameter. All we do now is to call *name_proc* followed by a name

```
name_length.call "A.K.Karthikeyan"
```

Hence *name_length* accepts my name as parameter, calculates its length and prints it. Hence this example shows that its possible to return a Proc from a function.

Multi Threading

Usually a program is read line by line and is executed step by step by the computer. At any given point of time the computer executes only one instruction[37]. When technology became advanced it became possible to execute many instructions at once, this process of doing many things at the same time is called multi processing or parallel processing.

Imagine that you are tasked with eating 5 pizzas. It would take a long time for you to do it. If you could bring your friends too, then you people can share the load. If you can form a group of 20 people, eating 5 pizzas becomes as easy as having a simple snack. The time required to complete the assigned task gets reduced drastically.

In your Ruby programing you can make the interpreter execute code in a parallel fashion. The process of executing code parallel is called multi threading. To show how multithreading works type the program below in text editor and execute it.

```
#!/usr/bin/ruby
# multithreading.rb

a = Thread.new{
      i = 1;
      while i<=10
            sleep(1)
            puts i
            i += 1
      end
}
puts "This code comes after thread"
a.join
```

Here is the programs output

```
This code comes after thread
1
2
3
4
5
6
7
8
```

37 Many computers now have more than one processing unit (aka core) and hence true miltithreading is possible now. Since you are a Ruby programmer there is no need for you to worry about it.

```
9
10
```

Unlike other programs this program will take 10 seconds to execute. Take a look at the program and output. The *puts "This code comes after thread"* comes after

```
a = Thread.new{
      i = 1;
      while i<=10
            sleep(1)
            puts i
            i += 1
      end
}
```

Yet it gets printed first. In the statements shown above we create a new thread named *a* in which we use a *while* loop to print from 1 to 10. Notice that we call a function called *sleep(1)* which makes the process sleep or remain idle for one second. A thread was created, while the thread is running, the Ruby interpreter checks the main thread and its comes across *puts "This code comes after thread"* and hence it prints out the string and in parallel as it executes the new thread *a* created by us, so 1 to 10 gets printed as we have inserted *sleep(1)* each loop takes about 1 second to execute. So after 10 seconds the thread *a* is finished.

The *a.join* tells the Ruby interpreter to wait till the thread finishes execution. Once the execution of thread *a* is over the statements after *a.join* (if any) gets executed. Since there is no statement after it the program terminates.

Here is another program that I took out from a website[38] that clearly explains about multithreading. Gothru the program carefully, try to understand how it works, I will explain it in a moment

```
#!/usr/bin/ruby
# multithreading_1.rb
def func1
   i=0
   while i<=2
      puts "func1 at: #{Time.now}"
      sleep(2)
      i=i+1
   end
end

def func2
   j=0
   while j<=2
```

38 If you think I have infringed a copyright, please inform me at ak@mindaslab.in or mindaslab@gmail.com

```
      puts "func2 at: #{Time.now}"
      sleep(1)
      j=j+1
   end
end

puts "Started At #{Time.now}"
t1=Thread.new{func1()}
t2=Thread.new{func2()}
t1.join
t2.join
puts "End at #{Time.now}"
```

Look at the highlighted code, we have created two threads *t1* and *t2*. Inside thread *t1* we call the method *func1()* and in thread *t2* we call the method *func2()*, by doing so we are executing both *func1()* and *func2()* in parallel. When executed this is how the output will look like[39]

```
Started At Sun Apr 25 09:37:51 +0530 2010
func1 at: Sun Apr 25 09:37:51 +0530 2010
func2 at: Sun Apr 25 09:37:51 +0530 2010
func2 at: Sun Apr 25 09:37:52 +0530 2010
func1 at: Sun Apr 25 09:37:53 +0530 2010
func2 at: Sun Apr 25 09:37:53 +0530 2010
func1 at: Sun Apr 25 09:37:55 +0530 2010
End at Sun Apr 25 09:37:57 +0530 2010
```

As you can see from the output, the outputs printed by *func1* and *func2* are interlaced which proves that they have been executed in parallel. Note that in func1 we have made the thread sleep for 2 seconds by giving *sleep(2)* and in *func2* we have made the thread sleep for 1 second by giving *sleep(1)*.

I have made small changes in *multithreading_1.rb* to produce *multithreading_2.rb* which gives almost the same result of multithreading_1.rb, so here is its code:

```
#!/usr/bin/ruby
# multithreading_2.rb

def func name, delay
   i=0
   while i<=2
      puts "#{name} #{Time.now}"
      sleep delay
      i=i+1
   end
end

puts "Started At #{Time.now}"
t1=Thread.new{func "Thread 1:", 2}
t2=Thread.new{func "Thread 2:", 3}
t1.join
t2.join
puts "End at #{Time.now}"
```

Instead of using two functions *func1* and *func2*, I have written a single function called *func* which

39 You may get a different output as you may be executing the program at a different time

accepts a *name* and time delay as input. A loop in it prints out the name passed and the time instant at which the statement gets executed. Notice the highlighted statements, we create two threads *t1* and *t2*. In thread *t1* I call the function *func* and pass along the name *Thread 1:* and tell it to sleep for 2 seconds. In thread *t2* I call the same *func* and pass name as *Thread 2:* and tell it to sleep for 3 seconds in each loop iteration. And when executed the program produces the following output

```
Started At Sun Apr 25 09:44:36 +0530 2010
Thread 1: Sun Apr 25 09:44:36 +0530 2010
Thread 2: Sun Apr 25 09:44:36 +0530 2010
Thread 1: Sun Apr 25 09:44:38 +0530 2010
Thread 2: Sun Apr 25 09:44:39 +0530 2010
Thread 1: Sun Apr 25 09:44:40 +0530 2010
Thread 2: Sun Apr 25 09:44:42 +0530 2010
End at Sun Apr 25 09:44:45 +0530 2010
```

Which very similar to output produced by *multithreading_1.rb*.

Scope of thread variables

A thread can access variables that are in the main process take the following program (*thread_variables.rb*) as example

```
#!/usr/bin/ruby
# thread_variables.rb

variable = 0
puts "Before thread variable = #{variable}"
a = Thread.new{
      variable = 5
}
a.join
puts "After thread variable = #{variable}"
```

Output

```
Before thread variable = 0
After thread variable = 5
```

Type the program and run it. It will produce the result shown above. As you can see from the program we initialize a variable named *variable* to *0* before we create the thread. Inside the thread we change the value of the *variable* to 5. After the thread block we print variables value which is now 5. This program shows us that you can access and manipulate a variable thats been declared in the main thread.

Lets now see if a variable created inside a thread can be accessed outside the scope of it? Type in the following program (*thread_variables_1.rb*) and execute it

```
#!/usr/bin/ruby
```

```
# thread_variables_1.rb

variable = 0
puts "Before thread variable = #{variable}"
a = Thread.new{
      variable = 5
      thread_variable = 72
      puts "Inside thread thread_variable = #{thread_variable}"
}
a.join
puts "================\nAfter thread\nvariable = #{variable}"
puts "thread_variable = #{thread_variable}"
```

Output

```
Before thread variable = 0
Inside thread thread_variable = 72
================
After thread
variable = 5
thread_variables_1.rb:13: undefined local variable or method `thread_variable'
for main:Object (NameError)
```

In the program above we see that we have created a variable named *thread_variable* in the thread *a*, now we try to access it in the following line:

```
puts "thread_variable = #{thread_variable}"
```

As you can see the output that the program / Ruby interpreter spits an error as shown:

```
thread_variables_1.rb:13: undefined local variable or method `thread_variable'
for main:Object (NameError)
```

It says there is an undefined local variable or method named *thread_variable*. This means that the statement in main thread is unable to access variable declared in the thread *a*. So from the previous two examples its clear that a thread can access a variable declared in the main thread whereas a variable declared in the thread's scope cannot be accessed by statement in main scope.

Thread Exclusion

Lets say that there are two threads that share a same resource, let the resource be a variable. Lets say that the first thread modifies the variable, while its modifying the second thread tries to access the variable, what will happen? The answer is simple and straight forward, though the program appears to run without errors you may not get the desired result. This concept is difficult to grasp, let me try to explain it with an example. Type and execute *thread_exclusion.rb*

```
#!/usr/bin/ruby
# thread_exclusion.rb

x = y = 0
diff = 0
Thread.new {
      loop do
```

```
            x+=1
            y+=1
      end
}
Thread.new {
      loop do
            diff += (x-y).abs
      end
}
sleep(1) # Here main thread is put to sleep
puts "difference = #{diff}"
```

Output

```
difference = 127524
```

Read the program carefully. Before we start any thread, we have three variables `x, y` and `diff` that are assigned to value `0`. Then in the first thread we start a loop in which we increment the value of `x` and `y`. In another thread we find the difference between `x` and `y` and save it in a variable called `diff`. In the first thread `x` and `y` are incremented simultaneously, hence the statement `diff += (x-y).abs` should add nothing to variable `diff` as `x` and `y` are always equal and their difference should always be zero, hence the absolute value of their difference will also be zero all the time.

In this program we don't wait for the threads to join (as they contain infinite loop), we make the main loop sleep for one second using the command sleep(1) and then we print the value of `diff` in the following statement

```
puts "difference = #{diff}"
```

One would expect the value to be zero but we got it as 127524, in your computer the value could be different as it depends on machine speed, what process its running and blah blah. But the moral is `diff` that should be zero has some value, how come?

We see in the first loop that `x` is incremented and then `y` is incremented, lets say that at an instant `x` value is 5 and `y` value is 4. `x` had just got incremented in the statement `x += 1` and now the Ruby interpreter is about to read and execute `y +=1` which will make y from 4 to 5. At this stage the second thread is executed by the computer. So in the statement

```
diff += (x-y).abs
```

putting `x` as 5 and `y` as 4 will mean that `diff` will get incremented by 1. In similar fashion while the main loop sleeps for one second, the two thread we have created would have finished thousands of loop cycles, hence the value of `diff` would increased significantly. Thats why we get the value of `diff` as a large number.

Well, we have seen how not to write the program in a wrong way, lets now see how to write it in the right way. Our task now is to synchronize the two threads that we have created so that one thread does not access other threads resources when the other is in middle of some busy process. To do so

we will be using a thing called Mutex which means Mutual Exclusion. Type the following program *thread_exclusion_1.rb* in text editor and execute it

```
#!/usr/bin/ruby
# thread_exclusion_1.rb

require 'thread'

mutex = Mutex.new
x = y = 0
diff = 0
Thread.new {
      loop do
            mutex.synchronize do
                  x+=1
                  y+=1
            end
      end
}
Thread.new {
      loop do
            mutex.synchronize do
                  diff += (x-y).abs
            end
      end
}
sleep(1) # Here main thread is put to sleep
puts "difference = #{diff}"
```

Output

```
difference = 0
```

As you see above, we get output as *difference = 0* , which means that *diff* variable is zero. Some thing prevented the second thread from accessing *x* and *y* in the first thread while the first one was busy. Some how the two threads have learned to share their resources. Study the code carefully, we see that we have included a thread package by typing

```
require 'thread'
```

Next we have created a Mutex variable named *mutex* using the following command

```
mutex = Mutex.new
```

Well then the code is as usual except inside the threads. Lets look into the first thread

```
Thread.new {
      loop do
            mutex.synchronize do
                  x+=1
                  y+=1
            end
      end
}
```

We see that the statements *x += 1* and *y += 1* are enclosed in *mutex.synchronize* block. In similar way the code computing the difference of *x* and *y* is also enclosed in *mutex.synchronize*

block as shown:

```
Thread.new {
      loop do
            mutex.synchronize do
                  diff += (x-y).abs
            end
      end
}
```

By doing so that we tell the computer that there is a common resource is shared by these two threads and one thread can access the resource only if other releases it. By doing so, when ever the difference (`diff`) is calculated in `diff += (x-y).abs` , the value of `x` and `y` will always be equal, hence `diff` never gets incremented and stays at zero forever.

Deadlocks

Have you ever stood in a queue,or waited for something. One place we all wait is in airport for our luggages to be scanned and cleared. Lets say that the luggage scanning machine gets out of order and you are stuck in airport. You are expected to attend an important company meeting and you have the key presentation and you must give an important talk. Since your baggage scanner failed the resource needed by you is not available to you, and you have the key knowledge to talk in the meeting and hence the meeting gets screwed up. One among the meeting might have promised to take his family to a movie, he might return late after the delayed meeting and hence all screw up.

Imagine that the baggage scanner, you, a person who must listen to your meeting are threads of a Ruby program. Since the baggage scanner wont release a resource (your bag) your process gets delayed and since you are delayed many other processes that depends on you are delayed. This situation is called dead lock. It happens in Ruby programs too.

When ever situation like this arises people wait rather than to rush forward. You wait for your bag to be released, your company waits for your arrival and the mans family waits for him to take them to movie. Instead of waiting if people rush up it will result in chaos.

Ruby has a mechanism to avoid this chaos and to handle this deadlock. We use a thing called condition variable. Take a look at the program (`thread_condition_variable.rb`) below, type it and execute it.

```
#!/usr/bin/ruby
# thread_condition_variable.rb

require 'thread'
```

```
mutex = Mutex.new

c = ConditionVariable.new
a = Thread.new {
   mutex.synchronize {
      puts "Thread a now waits for signal from thread b"
      c.wait(mutex)
      puts "a now has the power to use the resource"
   }
}

puts "(Now in thread b....)"

b = Thread.new {
   mutex.synchronize {
      puts "Thread b is using a resource needed by a, once its done it will
signal to a"
      sleep(4)
      c.signal
      puts "b Signaled to a to acquire resource"
   }
}
a.join
b.join
```

Output

```
Thread a now waits for signal from thread b
(Now in thread b....)
Thread b is using a resource needed by a, once its done it will signal to a
b Signaled to a to acquire resource
a now has the power to use the resource
```

Study the program and output carefully. Look at the output. First the statement in Thread *a* gets executed and it prints that thread *a* is waiting for Thread *b* to signal it to continue. See in thread *a* we have written the code *c.wait(mutex)* , where *c* is the Condition Variable declared in the code as follows:

```
c = ConditionVariable.new
```

See the highlighted code in *thread_condition_variable.rb* . So now thread *a* waits, now the execution focused on thread b when the following line is encountered in Thread *b*

```
puts "Thread b is using a resource needed by a, once its done it will signal to
a"
```

it prints out that thread b is using some resource needed by a, next thread b sleeps for 4 seconds because we have given *sleep(4)* in it. This sleep statement can be avoided, I have given a sleep because while the reader executes the program it make him wait and gives a feel that how really condition variable works.

Then after 4 seconds, Thread *b* signals to Thread *a* , its wait can end using the statement *c.signal* . Now Thread *a* receives the signal and can execute its rest of its statements, i.e after *c.signal* Thread *a* and Thread *b* can execute simultaneously.

Thread Exception

When ever there is an exception when the program is running, and if the exception isn't handled properly the program terminates. Lets see what happens when there is a exception in a thread. Type the code *thread_exception_true.rb* in text editor and execute it.

```
#!/usr/bin/ruby
# thread_exception_true.rb

t = Thread.new do
      i = 5
      while i >= -1
            sleep(1)
            puts 25 / i
            i -= 1
      end
end

t.abort_on_exception = true
sleep(10)
puts "Program completed"
```

Output

```
5
6
8
12
25
thread_exception_true.rb:8:in `/': divided by 0 (ZeroDivisionError)
      from thread_exception_true.rb:8
      from thread_exception_true.rb:4:in `initialize'
      from thread_exception_true.rb:4:in `new'
      from thread_exception_true.rb:4
```

Notice in the program we create a thread named *t*, and if you are quiet alert we haven't got *t.join* in the program. Instead of waiting for the thread to join we wait long enough for the thread to complete. For the thread to complete we wait for 10 seconds by using the statement *sleep(10)*.

Notice the line *t.abort_on_exception = true* where we set that if there raises an exception in the thread *t*, the program must abort. Lets now analyze whats in thread *t*. Thread *t* contains the following code

```
t = Thread.new do
      i = 5
      while i >= -1
            sleep(1)
            puts 25 / i
            i -= 1
      end
end
```

Notice that we divide 25 by *i* and put out the result of the division. *i* is decremented by 1 in each loop iteration, so when i becomes zero and when 25 is divided by it, it will raise an exception. So at

the sixth iteration of the loop, 25 is divided by zero, an exception is raised and the program stops by spiting out the following

```
thread_exception_true.rb:8:in `/': divided by 0 (ZeroDivisionError)
      from thread_exception_true.rb:8
      from thread_exception_true.rb:4:in `initialize'
      from thread_exception_true.rb:4:in `new'
      from thread_exception_true.rb:4
```

This happens because we have set *t.abort_on_exception* to *true* (see the highlighted code). What happens if we set it as *false*. Take a look at the program *thread_exception_false.rb*. In the program we have set *t.abort_on_exception* as *false*. Type the program in text editor and run it

```
#!/usr/bin/ruby
# thread_exception_false.rb

t = Thread.new do
      i = 5
      while i >= -1
            sleep(1)
            puts 25 / i
            i -= 1
      end
end

t.abort_on_exception = false
sleep(10)
puts "Program completed"
```

Take a look at output

```
5
6
8
12
25
Program completed
```

As you can see from the output there is no trace of exception occurrence[40]. The code that comes after thread *t* gets executed and we get output that says Program Completed. This is because we have set *abort_on_exception* to be *false*.

You can see from the last two programs that we haven't used *t.join* , instead we have waited long enough for the thread to terminate. This is so because once we join thread (that causes) exception with the parent (in this case the main) thread, the exception that arises in the child thread gets propagated to the parent / waiting thread so *abort_on_exception* has no effect even it set to *true* or *false*. So when ever exception raises it gets reflected on our terminal.

40 Exception has occurred but its not reported / aborted.

Thread Class Methods

There are certain thread methods which you can use to manipulate the properties of the thread. Those are listed below. If you can't understand a bit of it, never worry.

Sno.	Method	What it does
3.	`Thread.abort_on_exception`	Returns the status of the global *abort on exception* condition. The default is *false*. When set to *true*, will cause all threads to abort (the process will exit(0)) if an exception is raised in any thread.
4.	`Thread.abort_on_exception=`	When set to *true*, all threads will abort if an exception is raised. Returns the new state.
5.	`Thread.critical`	Returns the status of the global *thread critical* condition.
6.	`Thread.critical=`	Sets the status of the global *thread critical* condition and returns it. When set to *true*, prohibits scheduling of any existing thread. Does not block new threads from being created and run. Certain thread operations (such as stopping or killing a thread, sleeping in the current thread, and raising an exception) may cause a thread to be scheduled even when in a critical section.
7.	`Thread.current`	Returns the currently executing thread.
8.	`Thread.exit`	Terminates the currently running thread and schedules another thread to be run. If this thread is already marked to be killed, *exit* returns the *Thread.* If this is the main thread, or the last thread, exit the process.
9.	`Thread.fork { block }`	Synonym for Thread.new
10.	`Thread.kill( aThread )`	Causes the given *aThread* to exit
11.	`Thread.list`	Returns an array of *Thread* objects for all threads that are either runnable or stopped. Thread.
12.	`Thread.main`	Returns the main thread for the process.
13.	`Thread.new( [ arg ]* ) {| args | block }`	Creates a new thread to execute the instructions given in block, and begins running it. Any arguments passed to *Thread.new* are passed into the block.
14.	`Thread.pass`	Invokes the thread scheduler to pass execution to another thread.
15.	`Thread.start( [ args ] * ) {| args | block }`	Basically the same as *Thread.new* . However, if class *Thread* is subclassed, then calling *start* in that subclass will not invoke the subclass's *initialize* method.
16.	`Thread.stop`	Stops execution of the current thread, putting it into a *sleep* state, and schedules execution of another thread. Resets the *critical* condition to false.

Thread Instance Methods

Since everything in Ruby is an object, ia thread too is an object. Like many objects, threads have functions or methods which can be called to access or set a property in thread. Some of the functions and their uses are listed below (`thr` is an instance variable of the Thread class) :

Sno.	Method	What it does
1	`thr.alive?`	This method returns true if the thread is alive or sleeping. If the thread has been terminated it returns false.
2	`thr.exit`	Kills or exits the thread
3	`thr.join`	This process waits for the thread to join with the process or thread that created the child thread. Once the child thread has finished execution, the main thread executes the statement after thr.join
4	`thr.kill`	Same ad thr.exit
5	`thr.priority`	Gets the priority of the thread.
6	`thr.priority=`	Sets the priority of the thread. Higher the priority, higher preference will be given to the thread having higher number.
7	`thr.raise( anException )`	Raises an exception from *thr*. The caller does not have to be *thr*.
8	`thr.run`	Wakes up *thr*, making it eligible for scheduling. If not in a critical section, then invokes the scheduler.
10	`thr.wakeup`	Marks *thr* as eligible for scheduling, it may still remain blocked on I/O, however.
11	`thr.status`	Returns the status of *thr*: *sleep* if *thr* is sleeping or waiting on I/O, *run* if *thr* is executing, false if *thr* terminated normally, and *nil* if *thr* terminated with an exception.
12	`thr.stop?`	Waits for thr to complete via *Thread.join* and returns its value.
13	`thr[ aSymbol ]`	Attribute Reference - Returns the value of a thread-local variable, using either a symbol or a *aSymbol* name. If the specified variable does not exist, returns *nil*.
14	`thr[ aSymbol ] =`	Attribute Assignment - Sets or creates the value of a thread-local variable, using either a symbol or a string.
15	`thr.abort_on_exception`	Returns the status of the *abort on exception* condition for *thr*. The default is *false*.
16	`thr.abort_on_exception=`	When set to *true*, causes all threads (including the main program) to abort if an exception is raised in *thr*. The process will effectively *exit(0)*.

Regular Expressions

You are with your girl friend in a jewel shop, she chooses one of the est diamond rings and gives you THAT LOOK... Sure you know that you will fall into more credit card debt. You are in another talk with your boss to get more salary when he stares at you. THAT LOOK. You now know that what ever thing you say that you did good to the company, it will be ignored and would be futile. We all see for expressions in others face and we try to predict what next from it.

Lets say that you are on chat with your friend and he types :-) , you know he is happy, this :-(means he's sad. So its quiet evident that we can see expressions even in textual data, just as we see in each others face. Ruby's regular expression provides you with a way to detect these patterns in a given text and extract them if you wish. This can be used for some thing good. So this chapter is about that. What Ruby does not do is to tell you how to impress your girl without getting into debt :-(. Wish to report this as a bug and hope they will fix it in ruby 2.x ;-)

So fire up your irb, lets begin

Creating a empty regexp

Okay we will try to create a empty regular expression. In your terminal type `irb --simple-prompt` and in it type the following (without the `>>` , which is irb's prompt)

```
>> //.class
 => Regexp
```

You see //.class is Regexp. In other words any thing between those `/` and `/` is a regexp[41] . Regexp is not a string, but it denotes a pattern. Any string can match or may not match the pattern.

Detecting Patterns

Lets now see how to detect patterns in a regular expressions. Lets say that you want to see that abc is in a given string. Just punch the code below

```
>> /abc/.match "This string contains abc"
 => #<MatchData "abc">
```

abc is present in the given string hence you get a match. In the code snippet below, there is not abc

41 Will be using regexp instead of Regular expression from no on

in the string and hence it returns nil.

```
>> /abc/.match "This string contains cba"
 => nil
```

You can use the match function on a regexp as shown above or u can use it on a string, as shown below. Both gives you the same result.

```
>> "This string contains abc".match(/abc/)
=> #<MatchData "abc">
```

Another way of matching is shown below. You can use the =~ operator.

```
>> "This string contains abc" =~ /abc/
=> 21
 >> /abc/ =~ "This string doesn't have abc"
 => 25
```

The =~ tells you the location where the first occurrence of the match occurs.

Things to remember

There are some things you need to remember, or at least refer from time to time. Those are mentioned in table below[42].

Thing	What it means
.	Any single character
\w	Any word character (letter, number, underscore)
\W	Any non-word character
\d	Any digit
\D	Any non-digit
\s	Any whitespace character
\S	Any non-whitespace character
\b	Any word boundary character
^	Start of line
$	End of line
\A	Start of string
\z	End of string
[abc]	A single character of: a, b or c
[^abc]	Any single character except: a, b, or c
[a-z]	Any single character in the range a-z
[a-zA-Z]	Any single character in the range a-z or A-Z

42 I got this list from http://rubular.com/

Thing	What it means
`(...)`	Capture everything enclosed
`(a|b)`	a or b
`a?`	Zero or one of a
`a*`	Zero or more of a
`a+`	One or more of a
`a{3}`	Exactly 3 of a
`a{3,}`	3 or more of a
`a{3,6}`	Between 3 and 6 of a
`i`	case insensitive
`m`	make dot match newlines
`x`	ignore whitespace in regex
`o`	perform #{...} substitutions only once

Don't panic if you don't understand this, as I myself know nothing of it.

The dot

The dot in a reguar expression matches any thing,. To illustrate it let try some examples in our irb

```
>> /.at/.match "There is rat in my house"
=> #<MatchData "rat">
```

in the above code snippet, we try to match `/.at/` with a string, and it matches. The reason, the string contains a word called rat. Take a look at another two examples below, the `/.at/` matches cat and bat without any fuss.

```
>> /.at/.match "There is cat in my house"
=> #<MatchData "cat">

>> /.at/.match "There is bat in my house"
=> #<MatchData "bat">
```

Its not a rule that the dot should be at the start of the word or something, it can be any where. A regexp `/f.y/` could comfortably match fry and fly.

Character classes

Lets say tat we want to find that weather there is a bat or a rat or a cat is present in a given string. If there I we will print that there is a animal in the house,else we won't print it. You might be thinking e need to have three regexp like `/bat/` , `/cat/` and `/rat/` , but thats not the case. We know from those three regexp that only the first character varies. So what about /.at/ like the previous one. Well

that won't work either. Because if you eat too much and don't exercise well, that would match even fat …. :-))

So this time strictly e want to match only bat rat and cat, so we come up with a regexp like this: /[bcr]at/ , this will only match those three animal words and nothing else. Punch in the following example and run it in your computer

```
#!/usr/bin/ruby
# regexp_character_classes.rb

puts "There is a animal in your house"  if /[bcr]at/.match "There is bat in my
house"
puts "There is a animal in your house"  if /[bcr]at/.match "There is rat in my
house"
puts "There is a animal in your house"  if /[bcr]at/.match "There is cat in my
house"
puts "There is a animal in your house"  if /[bcr]at/.match "There is mat in my
house"
```

Output

```
There is a animal in your house
There is a animal in your house
There is a animal in your house
```

As you can see from the output above, the string “There is a animal in your house” gets printed thrice and not the fourth time where the match fails for "There is mat in my house".

Character classes can also accept ranges. Punch in the code below and run it.

```
#!/usr/bin/ruby
# regexp_character_classes_1.rb

print "Enter a short string: "
string = gets.chop
puts "The string contains character(s) from a to z"  if /[a-z]/.match string
puts "The string contains character(s) from A to Z"  if /[A-Z]/.match string
puts "The string contains number(s) from 0 to 9"  if /[0-9]/.match string
puts "The string contains vowels"  if /[aeiou]/.match string
puts "The string contains character(s) other than a to z"  if /[^a-z]/.match
string
puts "The string contains character(s) other than A to Z"  if /[^A-Z]/.match
string
puts "The string contains number(s) other than 0 to 9"  if /[^0-9]/.match string
puts "The string contains characters other than vowels"  if /[^aeiou]/.match
string
```

Output

```
Enter a short string: fly
The string contains character(s) from a to z
The string contains character(s) other than A to Z
The string contains number(s) other than 0 to 9
The string contains characters other than vowels
```

Output 1

```
Enter a short string: Burgundy 32
The string contains character(s) from a to z
The string contains character(s) from A to Z
The string contains number(s) from 0 to 9
The string contains vowels
The string contains character(s) other than a to z
The string contains character(s) other than A to Z
The string contains number(s) other than 0 to 9
The string contains characters other than vowels
```

OK, what you infer from the output? When you give fly these regexp's match:

- `/[a-z]/` since it contains a character from a to z
- `/[^A-Z]/` since it contains a character that does not belong any where from A to Z, hence you come to know `^` inside a capture means negation. There are also other uses for `^` which if I am not lazy you will be writing about it.
- `/[^0-9]/` since it does not contain any numbers from 0 to 9
- `/[^aeiou]/` since it does not contain a vowel (a or e or I or o or u)

According to that the messages in the puts statements gets printed.

Now look at the Output 1, I have given the program a string Burgundy 27 , check if your assumptions / logic tally with it ;-)

Scanning

I love this scan method in String Class. It lets us search a huge array of string for something. Just like needle in a haystack, since computers are getting faster and faster you can scan more and more. They are good for searching. They are quiet unlike the police in India, who would only conduct a search only if the person who has been burgled gives bribe.

So punch in the program below. It scans for words in a string.

```
#!/usr/bin/ruby
# regexp_scan.rb

string = """ There are some words in this string and this program will
scan those words and tell their word count """

words = string.scan(/\w+/)
puts "The words are:"
p words
puts # prints a empty line
puts "there are #{words.count} words in the string"
puts "there are #{words.uniq.count} unique words in string"
```

Output

```
The words are:
["There", "are", "some", "words", "in", "this", "string", "and", "this",
"program", "will", "scan", "those", "words", "and", "tell", "their", "word",
"count"]

there are 19 words in the string
there are 16 unique words in string
```

Note the */\w+/* what does it mean. Refer this table by clicking here. You can see that *\w* means any character like letter, number, underscore and *+* mean one or many. In other words I have assumed that words consists of any letter, number and underscore combinations and a word contains atlest one letter or more than that. So the statement *string.scan(**/\w+/**)* will scan all words and put into into a variable called *words* which we use in this program.

The statement p words prints out the array and in the following line

```
puts "there are #{words.count} words in the string"
```

we are counting the number of elements in the array words using the command *word.count* and embedding in a string using *#{words.count}* and printing it out to the user.

In the next statement

```
puts "there are #{words.uniq.count} unique words in string"
```

we are finding how many unique words are there in the given string using *words.uniq.count* and printing that too to the user. For example if you scan a large book of two authors and feed it to this program, the person who has more number of unique words can be assumed to have better vocabulary.

Lets now see another program. For example take a tweet you will do on twitter and you want to find out if the tweet contains twitter user names. So lets analyze a twitter user name, it first contains an @ symbol followed by a word. In other words it must match the regexp /@\w+/ , well simple. In the following program we scan all the users mentioned in a tweet

```
#!/usr/bin/ruby
# regexp_scan_twitter_users.rb

string = """ There is a person @karthik_ak who wrote ilr. Its about a
language called @ruby invented by @yukihiro_matz """

users = string.scan(/@\w+/)
puts "The users are:"
p users
```

Output

```
The users are:
["@karthik_ak", "@ruby", "@yukihiro_matz"]
```

Notice the highlighted statement in the program above. It scans all words that is perpended with an @ symbol, collects them and returns them as an array. Ans we just display that array using `p users`.

Captures

We have seen how useful regular expressions could be. Now let say we found a match with a regular expression and we just want to capture a small part of it, say a user name in a email, or a moth in some ones date of birth, how to do that?

We use square brackets for that and we call them captures (technically). Below is a program that asks birthday of a person and extracts the month out of it.

```
#!/usr/bin/ruby
# regexp_capture.rb

print "Enter Birthday (YYYY-MM-DD) :"
date = gets.chop
/\d{4}-(\d{2})-\d{2}/.match date
puts "You were born on month: #{$1}"
```

Output

```
Enter Birthday (YYYY-MM-DD) :1982-11-22
You were born on month: 11
```

Notice this line `/\d{4}-(\d{2})-\d{2}/.match` date here we check if the date matches the following. That is it must have four digits `/\d{4}/` , then it must be followed by a hyphen `/\d{4}-/` then it must be followed by two digits /\d{4}-\d(2}/ and it must be followed a hyphen and another two digits `/\d{4}-\d{2}-\d{2}/` .

Now we need to capture just the month thats in the middle. Hence we put braces around it like shown `/\d{4}-(\d{2})-\d{2}/` , we stick this regexp in the program above, in this line

```
/\d{4}-(\d{2})-\d{2}/.match date
```

Now where this capture `(\d{2})` gets stored. It gets stored in a global variable `$1`, if there is another capture, it gets stored in another variable `$2`, `$3` and so on..... So we now know `$1` has the month and we use it in the following line to print out the result

```
puts "You were born on month: #{$1}"
```

In the coming example `regexp_capture_1.rb` , we tr three captures where we want to capture

Year, Month and Date in one go. Hence we use the following regexp */(\d{4})-(\d{2})-(\d{2})/*. Type the program below and execute it.

```
#!/usr/bin/ruby
# regexp_capture_1.rb

print "Enter Birthday (YYYY-MM-DD) :"
date = gets.chop
/(\d{4})-(\d{2})-(\d{2})/.match date
puts "Year: #{$1} \n Month: #{$2} \n Date: #{$3}"
```

Output

```
Enter Birthday (YYYY-DD-MM) :1997-12-67
Year: 1997
 Month: 12
 Date: 67
```

Here the first capture starting from left is stored in *$1*, the second is stored in *$2* and the third in *$3* and so on (if we had given more). If you are wondering what *$0* is, why don't you give a puts *$0* statement at the end of the program and see what happens?

In the next program, we have designed it to tolerate some errors in user input. The user may not always give *1990-04-17* , he might give it as say *1990 - 04- 17* or some thing like that that might have spaces around numbers. Type int the program and execute it

```
#!/usr/bin/ruby
# regexp_capture_2.rb

print "Enter Birthday (YYYY-MM-DD) :"
date = gets.chop
/\s*(\d{4})\s*-\s*(\d{2})\s*-\s*(\d{2})\s*/.match date
puts "Year: #{$1} \n Month: #{$2} \n Date: #{$3}"
```

Output

```
Enter Birthday (YYYY-MM-DD) :1947- 07 - 14
Year: 1947
 Month: 07
 Date: 14
```

As you can see, the program rightly finds month, date and year! Well if you note the regexp we are using */\s*(\d{4})\s*-\s*(\d{2})\s*-\s*(\d{2})\s*/* we have padded digits with *\s**, now whats that. Once again refer the regexp table by clicking here. \s means space and * means zero or more, so say *\s*\d{4}* means match in such a way that the regexp has 4 digits and is perpended with zero or more spaces and *\s*\d{4}\s** match a 4 digit number which is perpended and followed by zero or more spaces. Hence no matter how much padding you give with space, it rightly finds out the dates.

Nested Capture

Sometimes some wise crack programmer just out of sheer luncticity[43] or just to make fun of your regexp skills or to challenge you, or for some other reason might perform a nested capture. That is a capture within a capture.

I haven't done it so far my self, but wish to do it in practical applications for the fun of it. Now I have worked out a nested capture example here on irb and if you like try it yourself

```
>> /(a(c(b)))/.match "This sting contains acb ao it has match"
=> #<MatchData "acb" 1:"acb" 2:"cb" 3:"b">
>> $1
=> "acb"
>> $2
=> "cb"
>> $3
=> "b"
```

See the highlighted part, the regexp is like this `/(a(c(b)))/` , does it makes any sense to you? Well then how to read it. First remove all brackets and read it like `/acb/` . `acb` is present in the string and hence its matched so we get the highlighted part in output as shown below

```
=> #<MatchData "acb" 1:"acb" 2:"cb" 3:"b">
```

Now apply the outer most bracket from left and we get a capture as shown `/(acb)/` , this matches and captures acb which appears in highlighted part in output as shown below

```
=> #<MatchData "acb" 1:"acb" 2:"cb" 3:"b">
```

This capture s stored in $1 global variable. Now forge the outer bracket and move from left to the second pair of brackets and you get the following regexp `/a(cb)/` , this matches `acb` and captures `cb` in the string, this is caught in variable `$2` and is also shown in highlighted part of the matched data below

```
=> #<MatchData "acb" 1:"acb" 2:"cb" 3:"b">
```

In similar way the inner most bracket pair, forms this regexp `/ac(b)/` and its captured in variable `$3` is showed in matched output below

```
=> #<MatchData "acb" 1:"acb" 2:"cb" 3:"b">
```

43 I like to create new english words :-))

Anchors and Assertions

Anchors

Anchors are reference points in Ruby. Lets say that we want to check if a line immediately begins with a `=begin`[44] , then I can check it wit a regexp like this `/^=begin/` , where the `^` sign is a anchor that represents beginning of the line:

```
/^=begin/.match "=begin"
=> #<MatchData "=begin">
```

Lets say like we have multiple line string and we want to extract a line (the first one). So the content of the first line could be any thing, so we get a regexp as `/.+/`, now it must be between beginning of line `^` and end of line `$` , so we get a regexp as shown `/^.+$/` , s this will match anything thats between line anchors. An example is shown below.

```
>> /^.+$/.match "Boom \n Thata"
=> #<MatchData "Boom ">
```

In the above example, note that \n splits the string into two lines as `\n` stands for newline character. So the regexp faithfully matches the first line content, that is “Boom ”.

The next type of Anchors we have are `\A` that stands for start of a string and `\z` that stands for end of a string. In the example below, we check if the string starts with some thing by using the following regexp `/\ASomething/`

```
>> /\ASomething/.match "Something is better than nothing"
=> #<MatchData "Something">
```

And we get a match. In the example below we get a nil match because the string does not start with Something.

```
>> /\ASomething/.match "Everybody says Something is better than nothing"
=> nil
```

Now lets check if nothing is followed by end of string, hence we form a regexp as shown `/nothing\z/`

```
>> /nothing\z/.match "Everybody says Something is better than nothing"
=> #<MatchData "nothing">
```

As expected we get a match of nothing. One should not that anchors will not be reflected in match data, anchor is not a character, but just a symbolic representation of position. So if you are expecting a match of nothing\z , forget it.

44 =begin represents start of block comment in Ruby, but it must start at the line beginning

```
>> /nothing\z/.match "Everybody says Something is better than nothing\n"
=> nil
```

Look at the example above, the `\z` matches a string without a line ending(`\n`) character. If you want to check for line endings, you must use capital Z like the example shown below.

```
>> /nothing\Z/.match "Everybody says Something is better than nothing\n"
=> #<MatchData "nothing">
```

So :-) it matches!

In the example below, we match all the stuff thats in a string with \n as its ending.

```
>> /\A.+\Z/.match "Everybody says Something is better than nothing\n"
=> #<MatchData "Everybody says Something is better than nothing">
```

Assertions

Lets say that you are searching for this man David Copperfield, perhaps to make your nagging girl friend or wife to disappear so that you can get a new one. You have a huge directory of names and you want to match his name. We can do those kind of matches using assertions[45] . So you want to search something that comes before Copperfield, for that we use look ahead assertions. Take the example below

```
>> /\w+\s+(?=Copperfield)/.match "David Copperfield"
=> #<MatchData "David ">
```

Look at the code thats been made bold. It has (?=Copperfield) , that is its looking forward for something, this time its Copperfield. Want to become rich soon, then search for (?=Goldfield) and want some good music, search for (?=Oldfield). Okay I am typing too much.

Here is the thing, what ever you give between (?= and) will be look forward and if there is something before it, it will be matched. So there is David before Copperfield, hence it was matched. Note that I had given \w+\s+ which means that I want to match a regexp of one or more letters, followed by one or more spaced that precedes before Copperfield. So here we have another example, that gives a positive match:

```
>> /\w+\s+(?=Copperfield)/.match "Joan Copperfield"
=> #<MatchData "Joan ">
```

Lets say that we want to match all those names which does not end with Copperfield, we will use look ahead,, negative assetion. For this we put Copperfield in between (?! and), so in the following example it will return a negative match

45 If you are thinking that this can be searched in much simple way, then ur right, but for the time being don't think too much. Your brain might overheat and burn.

```
>> /\w+\s+(?!Copperfield)/.match "Joan Copperfield"
=> nil
```

But in the next example it will return a positive match, because Joan is not before Copperfield

```
>> /\w+\s+(?!Copperfield)/.match "Joan Kansamy"
=> #<MatchData "Joan ">
```

We have seen look forward assertion. Now lets look at like backward assertion. Lets say that we wan to match last ame of person who's first name is David. Then we can look backwards from last name and see if its David. Checkout the code below

```
>> /(?<=David)\s+\w+/.match "Joan Kansamy"
=> nil
```

See the highlighted part. We have put David between (?<= and), so thats how you specify look back assertion. The above code returns nil because we have no David in it.

```
>> /(?<=David)\s+\w+/.match "David Munsamy"
=> #<MatchData " Munsamy">
```

The above example matches " Munsamy"[46] because we have David before the pattern \s+\w+

Same way like we had negative look forward, why can't we have it in look backwards. Jut replace = with a ! and you will get a negative assertion. So the example below will not match because you have David in front of Munsamy.

```
>> /(?<!David)\s+\w+/.match "David Munsamy"
=> nil
```

Now take this example. We will have a match. Because there is no David in front of the first \w+\s+ , that is in the example below it is a space followed by "in".

```
>> /(?<!David)\s+\w+/.match "All in all Munsamy"
=> #<MatchData " in">
```

46 Notice the space too which has been matched.

Final Word

For 2012 Edition

I never expected that I will be adding another touch to I Love Ruby. I am now working in a company called Webtoday Business, here I have 3 people under me who are creating web apps using Ruby on Rails. I recommended some books for them to learn Ruby, but when I gave ilr to them they said it was good and must be upgraded, so I started the process of touching the old book to give new finish.

I have added Regular Expressions chapter in this book. This chapter is far from exhaustive. From now on all ilr releases will be in relatively quick succession, with little increments and improvements to it. They may never ever be a major release.

Any way if you have been benefited by this book, drop a word to me.

For 2010 Edition

This is my first book on Ruby. I have just one person for help to write this book, hence I infer there could be many mistakes that could irritate the reader. I would be happy if the reader forgives me. Happier still will be I if the reader spots mistakes and notify my. It helps me perfect this book. I hope this book one day will be the one of the cost effective and quality Ruby study material available on this planet. Practice a lot, don't just limit with just one study material for Ruby. With confidence and hard wok I am sure you'll be one of the best Ruby programmers ever. Bye!

Appendix

An important Math Discovery

I must confess, sometimes computers do go wrong. In certain cases 1 is equal to 2. Take the equation $a^2-b^2=(a+b)(a-b)$, now put a = b = 1 in it so we get

$$1^2-1^2=(1+1)(1-1)$$
$$1-1=(1+1)(1-1)$$
$$\frac{1-1}{1-1}=1+1$$

Lets put $1-1=k$ so we get

$$\frac{k}{k}=1+1$$

Any number divided by itself is one, so $\frac{k}{k}=1$, so we get the proof that

$$\frac{k}{k}=1=1+1$$
$$1=1+1$$
$$1=2$$

Didn't I say that computers can go wrong[47].

47 Tell this important great discovery to no one. Is a great secret many mathematicians have failed to find out!

Sponsors

- Ruby is easy to learn and is a simple programming language
- Ruby is a free software
- Knowledge of Ruby is essential for Rails (a web development framework)
- This is a free book. Free here stands for free as in freedom, not free beer
- Contains working examples
- You can share this book with any one, copy it, distribute it, make any changes, I won't sue you. Promise!!
- Tweet your feedbacks to @karthik_ak , possibly with tag #ilr
- Please try visiting http://fsf.org and learn about free software, learn and contribute to it!